

c o n f e r e n c e

.....  
*p r o c e e d i n g s*

**FREENIX Track**  
**2003 USENIX Annual**  
**Technical Conference**

*San Antonio, Texas, USA*

*June 9–14, 2003*

Sponsored by  
The **USENIX** Association

**USENIX**  
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION



For additional copies of these proceedings contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA  
Phone: 510 528 8649  
FAX: 510 548 5738  
Email: [orders@usenix.org](mailto:orders@usenix.org)  
Web: <http://www.usenix.org>

The price is \$30 for members and \$40 for nonmembers.  
Outside the U.S.A. and Canada, please add  
\$18 per copy for postage (via air printed matter).

### **Past FREENIX Proceedings**

FREENIX '02	2002	Monterey, CA	\$22/30
FREENIX '01	2001	Boston, MA	\$22/30
FREENIX '00	2000	San Diego, CA	\$22/30
FREENIX '99	1999	Monterey, CA	\$22/30

© 2003 by The USENIX Association  
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-11-0

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

**USENIX Association**

**Proceedings of the  
FREENIX Track**

**2003 USENIX Annual Technical Conference**

**June 9–14, 2003  
San Antonio, Texas, USA**

## **Program Organizers**

### **Program Chair**

Erez Zadok, *Stony Brook University*

### **Program Committee**

David Beazley, *The University of Chicago*

Ray Bryant, *SGI*

Chuck Cranor, *AT&T Labs—Research*

Angelos Keromytis, *Columbia University*

Chuck Lever, *Network Appliance*

Bart Massey, *Portland State University*

Keith Packard, *HP Cambridge Research Labs*

Guido van Rooij, *Madison Gurkha*

Robert Watson, *Network Associates Laboratories &  
The FreeBSD Project*

Carl Worth, *USC, Information Sciences Institute*

### **The USENIX Association Staff**

## **External Reviewers**

Kostas G. Anagnostakis

Michael Eisler

Sotiris Ioannidis

Scott Long

Stefan Miltchev

Richard Offer

Colin Perkins

Ethan Solomita

Charles P. Wright

# 2003 USENIX Annual Technical Conference

## FREENIX Track

June 9–14, 2003

San Antonio, Texas, USA

Index of Authors .....	vii
------------------------	-----

Message from the Program Chair .....	viii
--------------------------------------	------

### Thursday, June 12, 2003

#### Network Services

*Session Chair: Robert Watson, Network Associates Laboratories & The FreeBSD Project*

Implementation of a Modern Web Search Engine Cluster .....	1
<i>Maxim Lifantsev and Tzi-cker Chiueh, Stony Brook University</i>	

CSE—A C++ Servlet Environment for High-Performance Web Applications .....	15
<i>Thomas Gschwind and Benjamin A. Schmit, Technische Universität Wien</i>	

U-P2P: A Peer-to-Peer Framework for Universal Resource Sharing and Discovery .....	29
<i>Neal Arthorne, Babak Esfandiari, and Alope Mukherjee, Carleton University</i>	

#### Mail

*Session Chair: Carl Worth, University of Southern California*

GNU Mailman, Internationalized .....	39
<i>Barry A. Warsaw, Zope Corporation</i>	

ASK: Active Spam Killer .....	51
<i>Marco Paganini</i>	

Learning Spam: Simple Techniques for Freely-Available Software .....	63
<i>Bart Massey, Mick Thomure, Raya Budrevich, and Scott Long, Portland State University</i>	

#### Network Protocols

*Session Chair: David Beazley, University of Chicago*

Network Programming for the Rest of Us .....	77
<i>Glyph Lefkowitz, Twisted Matrix Labs; Itamar Shtull-Trauring, Zoteca</i>	

In-Place Rsync: File Synchronization for Mobile and Wireless Devices .....	91
<i>David Rasch and Randal Burns, Johns Hopkins University</i>	

NFS Tricks and Benchmarking Traps .....	101
<i>Daniel Ellard and Margo Seltzer, Harvard University</i>	

**Friday, June 13, 2003**

**BIOS and Virtual Devices**

*Session Chair: Guido van Rooij, Madison Gurkha*

Flexibility in ROM: A Stackable Open Source BIOS ..... 115

*Adam Agnew and Adam Sulmicki, University of Maryland at College Park; Ronald Minnich, Los Alamos National Lab; William Arbaugh, University of Maryland at College Park*

Console over Ethernet ..... 125

*Michael Kistler, Eric Van Hensbergen, and Freeman Rawson, IBM Austin Research Laboratory*

Implementing a Clonable Network Stack in the FreeBSD Kernel ..... 137

*Marko Zec, University of Zagreb*

**File Systems**

*Session Chair: Chuck Lever, Network Appliance*

StarFish: Highly-Available Block Storage ..... 151

*Eran Gabber, Jeff Fellin, Michael Flaster, Fengrui Gu, Bruce Hillyer, Wee Teck Ng, Banu Özden, and Elizabeth Shriver, Lucent Technologies—Bell Laboratories*

Secure and Flexible Global File Sharing ..... 165

*Stefan Miltchev, University of Pennsylvania; Vassilis Prevelakis, Drexel University; Sotiris Ioannidis, University of Pennsylvania; John Ioannidis, AT&T Labs—Research; Angelos D. Keromytis, Columbia University; Jonathan M. Smith, University of Pennsylvania*

The CryptoGraphic Disk Driver ..... 179

*Roland C. Dowdeswell, The NetBSD Project; John Ioannidis, AT&T Labs—Research*

**X Window System**

*Session Chair: Bart Massey, Portland State University*

xstroke: Full-screen Gesture Recognition for X ..... 187

*Carl D. Worth, University of Southern California*

Matchbox: Window Management Not for the Desktop ..... 197

*Matthew Allum, OpenedHand Ltd.*

X Window System Network Performance ..... 207

*Keith Packard and James Gettys, Cambridge Research Laboratory, HP Labs*

## Saturday, June 14, 2003

### Experiences

*Session Chair: Keith Packard, HP Cambridge Research Labs*

Building a Wireless Community Network in the Netherlands ..... 219  
*Rudi van Drunen, Dirk-Willem van Gulik, Jasper Koolhaas, Huub Schuurmans, and Marten Vijn, WirelessLeiden Foundation*

OpenCM: Early Experiences and Lessons Learned ..... 231  
*Jonathan S. Shapiro, John Vanderburgh, and Jack Lloyd, Johns Hopkins University*

Free Software and High-Power Rocketry: The Portland State Aerospace Society ..... 245  
*James Perkins, Andrew Greenberg, Jamey Sharp, David Cassard, and Bart Massey, Portland State University*

### Privilege Management

*Session Chair: Angelos D. Keromytis, Columbia University*

POSIX Access Control Lists on Linux ..... 259  
*Andreas Grünbacher, SuSE Linux AG*

Privman: A Library for Partitioning Applications ..... 273  
*Douglas Kilpatrick, Network Associates Laboratories*

The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0 ..... 285  
*Robert Watson, Wayne Morrison, and Chris Vance, Network Associates Laboratories; Brian Feldman, The FreeBSD Project*

### Kernel

*Session Chair: Ray Bryant, SGI*

Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel ..... 297  
*Andrea Arcangeli, SuSE Labs; Mingming Cao, Paul E. McKenney, and Dipankar Sarma, IBM Corporation*

An Implementation of User-level Restartable Atomic Sequences on the NetBSD Operating System ..... 311  
*Gregory McGarry*

Providing a Linux API on the Scalable K42 Kernel ..... 323  
*Jonathan Appavoo, University of Toronto; Marc Auslander, Dilma Da Silva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis, IBM T.J. Watson Research Center*



## Index of Authors

Agnew, Adam	115	McGarry, Gregory	311
Allum, Matthew	197	McKenney, Paul E.	297
Appavoo, Jonathan	323	Miltchev, Stefan	165
Arbaugh, William	115	Minnich, Ronald	115
Arcangeli, Andrea	297	Morrison, Wayne	285
Arthorne, Neal	29	Mukherjee, Alope	29
Auslander, Marc	323	Ng, Wee Teck	151
Budrevich, Raya	63	Ostrowski, Michal	323
Burns, Randal	91	Özden, Banu	151
Cao, Mingming	297	Packard, Keith	207
Cassard, David	245	Paganini, Marco	51
Chiueh, Tzi-cker	1	Perkins, James	245
Da Silva, Dilma	323	Prevelakis, Vassilis	165
Dowdeswell, Roland C.	179	Rasch, David	91
Edelsohn, David	323	Rawson, Freeman	125
Ellard, Daniel	101	Rosenburg, Bryan	323
Esfandiari, Babak	29	Sarma, Dipankar	297
Feldman, Brian	285	Schmit, Benjamin A.	15
Fellin, Jeff	151	Schuermans, Huub	219
Flaster, Michael	151	Seltzer, Margo	101
Gabber, Eran	151	Shapiro, Jonathan S.	231
Gettys, James	207	Sharp, Jamey	245
Greenberg, Andrew	245	Shriver, Elizabeth	151
Grünbacher, Andreas	259	Shtull-Trauring, Itamar	77
Gschwind, Thomas	15	Smith, Jonathan M.	165
Gu, Fengrui	151	Sulmicki, Adam	115
Hillyer, Bruce	151	Thomure, Mick	63
Ioannidis, John	165, 179	Vance, Chris	285
Ioannidis, Sotiris	165	Vanderburgh, John	231
Keromytis, Angelos D.	165	van Drunen, Rudi	219
Kilpatrick, Douglas	273	van Gulik, Dirk-Willem	219
Kistler, Michael	125	Van Hensbergen, Eric	125
Koolhaas, Jasper	219	Vijn, Marten	219
Krieger, Orran	323	Warsaw, Barry A.	39
Lefkowitz, Glyph	77	Watson, Robert	285
Lifantsev, Maxim	1	Wisniewski, Robert W.	323
Lloyd, Jack	231	Worth, Carl D.	187
Long, Scott	63	Xenidis, Jimi	323
Massey, Bart	63, 245	Zec, Marko	137



## Message from the Program Chair

Six years ago, the leaders of USENIX recognized the value of promoting Open Source Software (OSS). USENIX went on to create a FREENIX track in the Annual Technical Conference. FREENIX's main goal was to publish papers on various OSS topics, thus helping the community as a whole. Six years later, the OSS movement has changed a lot. It is now a very large movement, with numerous users and developers who all recognize the quality and value of OSS. Even large commercial vendors recognize this: they now bundle substantial OSS packages and even offer full support for OSS-based systems.

Establishing a new conference is a difficult task. In addition, FREENIX has special challenges. One of the main goals for FREENIX was to encourage submissions from OSS developers "in the trenches" who produce great software but are not accustomed to publishing papers. From its humble beginnings, FREENIX has always endeavored to keep improving each and every year. Thanks to all the hard work of those who organized FREENIX in past years, we were able this year to raise the bar even higher.

This year we received 70 submissions, a large increase from past years. The range of submissions really shows the maturity of FREENIX and the OSS movement. We received two-page extended abstracts as well as full-length 14-page papers; we got papers from students and professionals, from first-time authors and long-time researchers, from the USA as well as overseas, and covering traditional as well as new topics.

Since the page length of submissions as well as the experience of the authors vary considerably, we committed to review each paper thoroughly by as many people as possible. For two months after the submission due date last November, the Program Committee (PC) and additional external readers reviewed all 70 submissions. Each PC member read every paper, a total of 562 pages! Nine external readers read a substantial portion of the submissions. Together, we produced 874 reviews, a total of nearly 36,000 lines of text. No paper received fewer than ten reviews, and some got as many as fifteen.

The PC met in January for two days to select the 27 papers for this year's conference. In addition to the obvious goal of picking the best papers, we endeavored to pick influential papers—those that showcase new areas in OSS development, topics that highlight future directions, papers that will foster a lot of discussion, etc. Each paper was assigned a shepherd to help guide authors toward producing the final version of the paper. For nearly three months, authors and shepherds worked closely to produce the papers in these proceedings.

FREENIX is unique in that it is open to all OSS topics; thus it has something for everyone. We are quite proud of this year's papers. You will find papers on traditional topics such as the latest advances in Linux, \*BSD, and X11; you'll hear about new OSS topics such as search engines, extensible BIOSes, SANs, and micro-kernels; you'll get a hefty dose of current hot topics such as security, privacy, and anti-SPAM techniques; for the first time, a session entirely devoted to experiences with OSS; and yes, even rocket science!

A conference such as this could not happen without the help of many people. First, thanks go to the authors of all submitted papers. Next, I'd like to thank the program committee and the reviewers for the person-weeks they spent reviewing and shepherding papers. Special thanks go to the employers of all authors and reviewers, for giving their employees the time and support to write up their research during these difficult economic times. Finally, many thanks for the professional USENIX staff, a well-oiled machine, who toil endlessly behind the scenes and bring us several wonderful conferences each year.

**Erez Zadok, *Stony Brook University***  
**FREENIX Program Chair**

# Implementation of a Modern Web Search Engine Cluster

Maxim Lifantsev      Tzi-cker Chiueh

*Department of Computer Science*

*Stony Brook University*

maxim@cs.sunysb.edu

chiueh@cs.sunysb.edu

## Abstract

Yuntis is a fully-functional prototype of a complete web search engine with features comparable to those available in commercial-grade search engines. In particular, Yuntis supports page quality scoring based on global web linkage graph, extensively exploits text associated with links, computes pages' keywords and lists of similar pages of good quality, and provides a very flexible query language. This paper reports our experiences in the three-year development process of Yuntis, by presenting its design issues, software architecture, implementation details, and performance measurements.

## 1 Introduction

Internet-scale web search engines represent crucial web information access tools as well as pose software system design and implementation challenges that involve processing unprecedented volumes of data. To equip these search engines with sophisticated features compounds the overall architectural scale and complexity because this requires integration of non-trivial algorithms that can work efficiently with huge amounts of real-world data.

Yuntis is a prototype implementation of a scalable cluster-based web search engine that provides many modern search engine functionalities such as global linkage-based page scoring and relevance weighting [20], phrase extraction and indexing, and generation of keywords and lists of similar pages for all web pages. The entire Yuntis prototype consists of 167,000 lines of code, and represents a 3-man-year effort. In this paper, we discuss the design and implementation issues involved in the prototyping process of Yuntis. We intend this paper to shed some light into the internal workings of a feature-rich modern web search engine, and serve as a blueprint for future development of Yuntis.

The next two sections provide background and motivation for development of Yuntis, respectively. Section 4 describes its architecture. Implementation of main processing activities of Yuntis is covered in Section 5. Section 6 quantifies the performance of Yuntis. We conclude the paper by discussing future work in Section 7.

## 2 Background

The basic service offered by all web search engines is returning a set of web page URLs in response to a user query composed of words, thus providing a fast and hopefully accurate navigation service over the unstructured set of (interlinked) pages. To achieve this, a search engine must at least acquire and examine a set of URLs it wishes to provide searching capabilities for. This is usually done by fetching pages from individual web servers starting with some seed set, and then following the encountered links, while obeying some policy that limits and orders the set of examined pages. All fetched pages are preprocessed to later allow efficient answering of queries. This usually involves inverted word indexing, where for each encountered word the engine maintains the set of URLs the word occurs in (is relevant to), possibly along with other (positional) information regarding the individual occurrences of words. These indexes must be kept in a format that allows their fast intersection and merging during querying time, for example, they can be sorted in the same order by the contained URLs.

The contents of the examined pages can be kept so that relevant page fragments or whole pages can be also presented to users quickly. Frequently, some linkage-related indexes are also constructed, for instance, to answer queries about backlinks to a given page. Modern search engines following Google's example [5] can also associate with a page and index some text that is related to or contained in the links pointing to the page. With appropriate selection and weighing of such text fragments, the engine can leverage the page descriptions embedded into its incoming links.

Developing on the ideas of Page, et al [28] and Kleinberg [17], search engines now include some non-trivial methods of estimating the relevance or the "quality" of a web page for a given query using the linkage graph of the web. These methods can significantly improve the quality of search results, as evidenced by the search engine improvements pioneered by Google [11]. Here we consider only the methods for computing query-independent page quality or importance scores based on an iterative computation on the whole known web linkage graph [5, 19, 20, 28].

There are also other less widespread but useful search

engine functions, such as the following:

- Query spelling correction utilizing collected word frequencies.
- Understanding that certain words form phrases that can serve as more descriptive items than individual words.
- Determining most descriptive keywords for pages, which can be used for page clustering, classification, or advertisement targeting.
- Automatically clustering search results into different subgroups and appropriately naming them.
- Building high-quality lists of pages similar to a given one, thus allowing users to find out about alternatives to or analogs of a known site.

Several researchers have described their design and implementation experiences building different operations of large-scale web search engines, for example: The architecture of the Mercator web crawler is reported by Heydon and Najork [12]. Brin and Page [5] document many design details of the early Google search engine prototype. Design possibilities and tradeoffs for a repository of web pages are covered by Hirai, et al [13]. Bharat, et al [4] describe their experiences in building a fast web page linkage connectivity server. Different architectures for distributed inverted indexing schemes are discussed by Melnik, et al [24] and Ribeiro-Neto, et al [31].

In contrast, this paper primarily focuses on design and implementation details and considerations of a comprehensive and extensible search engine prototype that implements analogs or derivatives of many individual functions discussed in the mentioned papers, as well as several other features.

### 3 Motivation

Initially we wanted to experiment with our new model, the voting model [19, 20], for computing various “quality” scores of web pages based on overall linkage structure among web pages in the context of implementing web searching functions. We also planned to implement various extensions of the model that could utilize additional metadata for rating and categorizing web pages, for example, metadata parsed or mined from crawled pages or metadata from external sources such as the directory structure dumps for the Open Directory Project [27].

To do any of this, one needs the whole underlying system for crawling web pages, indexing their contents, doing other manipulations with the derived data, and finally presenting query results in a form appropriate for easy evaluation. The system must also be sufficiently scalable to support experiments with real datasets of considerable size, because page scoring algorithms

based on overall linkage in general produce better results when working with more data.

Since there was no reasonably scalable and complete web search engine implementation openly available that one could easily modify, extend, and experiment with, we needed to consider available subcomponents, and then design and build the whole prototype. Existing web search engine implementations were either trade secrets of the company that developed them, systems that were meant to handle small datasets on one workstation, or (non-open) research prototypes designed to experiment with some specific search engine technique.

## 4 Design of Yuntis

The main design goals of Yuntis were as follows:

- Scalability of data preparation at least to tens of millions of pages processed in a few days.
- Utilization of clusters of workstations for improving scalability.
- Faster development via simple architecture.
- Good extensibility for trying out new information retrieval algorithms and features.
- Query performance and flexibility adequate for quickly evaluating the quality of search results and investigating possible ways for improvement.

We chose C++ as the implementation language for almost all the functionality in Yuntis because it facilitates development without compromising efficiency. To attain decent manageability of the relatively large code-base we adopted the practice of introducing needed abstraction layers to enable aggressive code reuse. Templates, inline functions, multiple inheritance, and virtual functions all provide ways to do this, while still generating efficient code and getting as close to low-level bit manipulation as C when needed. We use abstract classes and inheritance to define interfaces and provide changeable implementations. Template classes are employed to reuse complex tasks and concepts. Although additional abstraction layers sometimes introduce run-time overheads, the reuse benefits were more important for building the prototype.

### 4.1 High-Level Yuntis Architecture

To maximize utilization of a cluster of PC workstations connected by a LAN, the Yuntis prototype is composed of several interacting processes running on the cluster nodes (see Figure 1). When an instance of the prototype is operational, each cluster node runs one database worker process that is responsible for storing, processing, and retrieving of all data assigned to the disks of that node. When needed, each node can also run one fetcher and one parser process that respectively retrieve and parse web pages that are stored on the corresponding



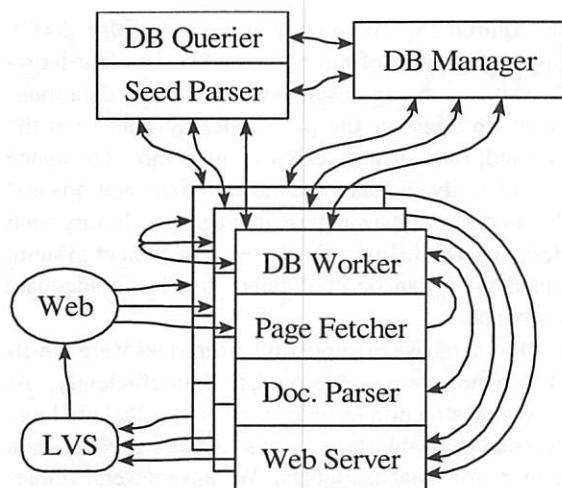


Figure 1: Yuntis cluster processes architecture.

node. There is one database manager process running at all times on one particular node. This process serves as the central control point, keeps track of all other Yuntis processes in the cluster, and helps them to connect to each other directly. Web servers answering user queries are run on several cluster nodes and are joined by the Linux Virtual Server load-balancer [23] into a single service.

There are also a few other auxiliary processes. The database querier helps with low-level manual examination and inspection of all the data managed by the database worker processes. Database rebuilders can initiate rebuilding of all data tables by feeding into the system the essential data from a set of existing data files. A seed data parsing and data dumping process can introduce initial data into the systems and extract some interesting data out of it.

A typical operation scenario of Yuntis involves starting up database manager and workers, importing an initial URL seed set or directory metadata, crawling from the seed URLs using fetchers and parsers, and complete preprocessing of the crawled dataset; then finally we start the web server process(es) answering user search queries. We discuss these stages in more detail in Section 5.

## 4.2 Library Building Blocks

We made major design decisions early in the development, that would later affect many aspects of the system. These decisions were about choosing the architecture for data storage, manipulation, and querying, as well as the approach to node-to-node cluster data communication. We also decided on the approach to interaction with web servers providing the web pages and with web clients querying the system. These activities capture all main processing in a search engine cluster. In addition, a pro-

cess model was chosen to integrate all these activities in one distributed system of interacting processes.

The choices about whether to employ or reuse code from an existing library or an application, or rather to implement the needed functionality afresh were made after assessing the suitability of existing code-bases and comparing the expected costs of both choices. Many of these choices were made without a comprehensive performance or architecture compatibility and suitability testing. Our informal evaluation deemed such costly testing not justified by the low expectation of its payoff to reveal a substantially more efficient design choice. For example, existing text or web page indexing libraries such as Isearch [15], ht://Dig [14], Swish [35], or Glimpse [37] were not designed to be a part of a distributed large-scale web search engine, hence the cost of redesigning and reusing them was comparable with writing own code.

### 4.2.1 Process Model

We needed an architecture that in one process space could simultaneously and efficiently support several of the following: high volumes of communication with other cluster nodes, large amounts of disk I/O, network communication with HTTP servers and clients, as well as significant mixing and exchange of data communicated in these ways. We also wanted to support multiple activities of each kind that individually need to wait for completion of some network, interprocess, or disk I/O.

To achieve this we chose an event-driven programming model that uses one primary thread of control that handles incoming events via a `select`-like data polling loop. We used this model for all processes in our prototype. The model avoids multi-threading overheads of task switching, stack allocation, and synchronization and locking complexities. But it also requires to introduce call/callback interfaces to all potentially blocking operations at all abstraction levels, from file and socket operations to exchanging data with a (remote) database table. Moreover, non-preemptiveness in this model requires us to ensure that processing of large data items can be split into smaller chunks so that the whole process can react to other events during such processing.

The event polling loop can be generalized to support interfaces with the operating system that are more efficient than, but similar to `select`, such as `Kqueue` [18]. We also later added support for fully asynchronous disk I/O operations via a pool of worker threads communicating through a pipe with the main thread.

Another reason for choosing the essentially uni-threaded event-driven architecture were the web server performance studies [16, 29] showing that web servers with such architecture under heavy loads significantly outperform web servers (such as Apache [2]) that al-

locate a process or a thread per each request. Hence Apache's code-base was not used as it has different process architecture and is targeted to support highly configurable web servers. Smaller `select`-based web servers such as `thttpd` [36] were designed to be just fast light-weight web servers without providing a more modular and extensible architecture. In our architecture, communication with HTTP servers and clients is handled by an extensible hierarchy of classes that in the end react to network socket and disk I/O events.

#### 4.2.2 Intra-Cluster Communication

We needed high efficiency of the communication for a specific application architecture instead of overall generality, flexibility, and interoperability with other applications and architectures. Thus we did not use existing network communication frameworks such as CORBA [8], SOAP [33], or DCOM [9] for communication among cluster workstations.

We did not employ network message-passing libraries such as MPI [25] or PVM [30] because they appear to be designed for scientific computing: they are oriented to support many tasks (frequently with multiprocessors in mind) that do not actively use local disks on the cluster workstations and do not communicate actively with many other network hosts. Because of inadequate communication calls, MPI and PVM require to use a lot of threads if one needs intensive communication. They do not have scalable primitives to simultaneously wait for many messages arriving from different points, as well as for readiness of disk I/O and other network I/O, for instance, over HTTP connections.

Consecutively, we developed our own cluster communication primitives. Information Service (IS) is a call/callback interface for a set of possibly remote procedures that can consume and produce small data items or long data streams. The data to be exchanged is untyped byte sequences and procedures are identified by integers. There is also an easy way to wrap this into a standard typed interface. We have implemented support for several IS clients and implementations to set up and communicate over a common TCP socket.

#### 4.2.3 Disk Data Storage

We did not use full-featured database systems mainly because the expected data and processing load required us to employ a distributed system running on a cluster of workstations and use light-weight data management primitives. We needed a data storage system with minimal processing and storage overheads oriented for optimizing the throughput of data-manipulation operations, not latency and atomicity of individual updates. Even high-end commercial databases appeared to not satisfy

these requirements completely at the time (May 2000). An indirect support of our choice is the fact that large-scale web search engines also use their own data management libraries for the page indexing data. On the other hand, our current design is quite modular, hence one could easily add database table implementations that could interface with a database management library such as Berkeley DB [3] or a database management system, provided these can be configured to achieve adequate performance.

A set of database manipulation primitives were developed to handle large-scale on-disk data efficiently. At the lowest abstraction level are virtual files that are large continuous growable byte arrays and are used as data containers for database tables. We have several implementations of the virtual file interface based on one or multiple physical files, memory-mapped file(s), or several memory regions. This unified interface allows the same database access code to run over physical files or memory regions.

The database table call/callback interface is at the next abstraction level, and defines a uniform interface to different kinds of database tables that share the same common set of operations: add, delete, read, or update (a part of) a record identified by a key. A database table implementation composed of disjoint subtables together with an interface to an Information Services instance allows a database table to be distributed across multiple cluster nodes while keeping data table's physical placement completely transparent to the code of its clients. To support safe concurrent accesses to a database table, we provide optional exclusive and shared locking at both the database record and database table levels.

At the highest abstraction level are classes and templates to define typed objects that are to be stored in database tables (or exchanged with Information Services), as well as to concisely write procedures that exchange information with database tables or IS'es via their call/callback interfaces. This abstraction level enables us to hide almost all the implementation details of the database tables behind a clean typed interface, at the cost of small additional run-time overheads. For example, we frequently read or write a whole data table record, when we are actually interested in just a few of its fields.

### 4.3 External Libraries and Tools

We have heavily relied on existing more basic and more compatible libraries and tools than the ones discussed earlier.

The Standard Template Library (STL) [34] of C++ proved to be very useful, but we had to modify it to enhance its memory management functionality by adding real memory deallocation, and eliminate a hash table

implementation inefficiency of erasing elements from a large, very sparse table.

GNU Nana library [26] is very convenient for logging and assertion checking during debugging, especially when the GNU debugger (GDB) [10] due to its own bugs often crashes while working with the core dumps generated by our processes. Consequently we had to rely more on logging and on attaching GDB to a running process, which consumes a fair amount of processing resources. Selective execution logging and extensive run-time assertion checking greatly helped in debugging our parallel distributed system.

The eXternalization Template Library [38] approach provides a clean, efficient, and extensible way to convert any typed C++ object into and from a byte sequence for compact transmission among processes on the cluster of workstations, or for long-term storage on disk.

Parallel compilation via the GNU make utility and simple scripts and makefiles, together with right granularity of individual object files, allowed us to reduce build times substantially by utilizing all our cluster nodes for compilation. For example, a full Yuntis build taking 38.9min for compilation and 2min for linking on one workstation takes 3.7+2min on 13 workstations.

## 4.4 Data Organization

We store information about the following kinds of objects: web hosts, URLs, web sites (which are sets of URLs most probably authored by the same entity), encountered words or phrases, and directory categories. All persistently stored data about these objects is presently organized into 121 different logical data tables. Each data table is split into partitions that are evenly distributed among the cluster nodes. The data tables are split into 60, 1020, 120, 2040, and 60 partitions for the data respectively related to one of the above five kinds of objects. These numbers are chosen as to ensure a manageable size of each partition for all data tables at the targeted size of a manipulated dataset.

All data tables (that is, their partitions) have one of the following structures: indexed array of fixed-sized records, array of fixed-sized records sorted by a field in each record, heap-like addressed set of variable-sized records, or queues of fixed- or variable-sized records. These structures cover all our present needs, but new data table structures can be introduced if needed. Records in all these structures except queues are randomly accessible by small fixed-sized keys. The system-wide keys for whole data tables contain a portion used to choose the partition and the rest of the key is used within the partition to locate a specific record (or a small set of matching records in the case of the sorted array structure).

For each of the above five kinds of web-world objects there are data tables to map between object names and internal identifiers which index fixed-sized information records, which in turn contain pointers into other tables with variable-sized information related to each object. This organization is both easy to work with and allows for a reasonably compact and efficient data representation.

The partition to store a data record is chosen by the hash values derived from the name of the object to which the record is most related. For example, if the hash value of a URL maps it to the  $i$ th partition out of 1020, then such items as the URL's name, the URL's information record, lists of back and forward links for the URL are all to be stored in the  $i$ th partition of the corresponding data table. One result of such data organization is that a database key or textual name of an object readily determines the database partition and cluster node the object belongs to. Hence, for all data accesses a database client can choose and communicate directly with the right worker without consulting any central lookup service.

## 4.5 Data Manipulation

The basic form of manipulation over data stored in the data tables is when individual data records or their parts are read or written by a local or remote request and the accessing client activity waits for completion of its request. There are two kinds of inefficiencies we would like to eliminate here: the network latency delay for remote accesses and local data access delays and overheads. The latter occur when the data needed to complete a data access has to be brought into memory from disk and into the CPU cache from memory. This can also involve the substantial processing overheads of working with data via file operations instead of accessing memory regions.

To avoid all these inefficiencies we rely on batched delayed execution of data manipulation operations—see Lifantsev and Chiueh [21] for full details. All large volume data reading (and update when possible) is organized around sequential reading of the data table partition files concurrently on all cluster nodes. In most other cases, when we need to perform a sequence of data accesses that work with remote or out-of-core data, we do not execute the sequence immediately. Instead we batch the needed initiation information into a queue associated with the group of related data table partitions this sequence of data accesses needs to work with. When such batching is done to a remote node, in most cases we do not need an immediate confirmation that the batching has completed in order to continue with our work. Thus most network communication delays are masked. After a large number of such initiation records are batched



to a given queue to justify the I/O costs (or when no other processing can proceed), we execute such a batch by loading or mapping into memory the needed data partitions and then working with the data in memory.

For many data tables, we can guarantee that each of their partitions will fit into the available memory, thus they are actually sequentially read from disk. For other data tables, the utilization of file mapping cache in the OS is significantly improved. With this approach, even for limited hardware resources, we can guarantee for a large spectrum of dataset sizes that in most cases all data manipulation happens with data already in local memory (or even CPU cache) via low-overhead memory access primitives. This model of processing utilizes such primitives as the following: support for database tables composed of disjoint partitions, buffered queues over several physical files for fast operation batching, classes to start and arbiter execution of operation batches and individual batched operations, and transparent memory-loading or mapping of selected database table partitions for the time of execution of an operations' batch.

In the end, execution of a batched operation consists of manipulating some database data already in memory and scheduling of other operations by batching their input data to an appropriate queue possibly on other cluster nodes. We wait for completion of this inter-node queueing only at batch boundaries. Hence, inter-node communication delays do not block execution of individual operations. High-level data processing tasks are organized by a controlling algorithm at the database manager process that initiates execution of appropriate operation batches and initial generation of operations. Both of these proceed on cluster nodes in parallel.

#### 4.5.1 Flow Control

During execution of operation batches (and operation generation by database table scanning) we need to have some flow control: On one hand, to increase CPU utilization, many operations should be allowed to execute in parallel in case some of them block on I/O. On the other hand, batch execution (sometimes even execution of a single operation) should be paused and resumed so that inter-cluster communication buffers are not needlessly large when they are being processed. Our adopted solution is to initiate a certain large number of operations in parallel and pause/resume their execution via appropriate checks/callbacks depending on the number of pending inter-cluster requests at this node. Allowing on the order of 100,000 pending inter-cluster requests appears to work fine for all Yuntis workloads. The exact number of operations potentially started in parallel is tuned depending on the nature of processing done by each class of operations and ranges from 20 to 20,000.

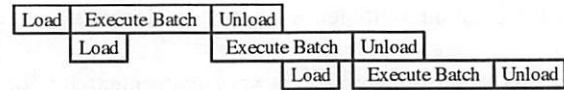


Figure 2: Operation batches execution pipeline.

#### 4.5.2 CPU and I/O Pipeline

Since most data processing is organized into execution of operation batches, we optimize it by scheduling it as a pipeline (see Figure 2). Each batch goes through three consecutive stages: reading/mapping of database partitions from disk, execution of its operations, and writing out of modified database data to disk. The middle stage is more CPU-intensive, while the other two are more I/O-intensive. We use two shared/exclusive locks and an associated sequence of operations with them to achieve pipeline-style exclusive/overlapped execution of CPU and I/O-intensive sections. This requires us to double the number of data partitions so that the data manipulated by two adjacent batches all fits into the available memory of a node.

### 5 Implementation of Yuntis

In the following sections we describe the implementation details and associated issues for the major processing activities of Yuntis mostly in the order of their execution. Table 1 provides a coarse breakdown for code sizes of major Yuntis subsystems.

#### 5.1 Starting Components Up

First, the database manager process is started up on some cluster node and begins listening on a designated TCP port. After that, the database worker processes are started on all nodes and start listening on another designated TCP port for potential clients, as well as advertise their presence to the manager by connecting to it. As soon as the manager knows that all workers are up, it sends the information about the host and port numbers of all workers to each worker. At this point each worker establishes direct TCP connections with all other workers and reports complete readiness to the manager.

Other processes are connected to the system in a similar fashion. A process first connects to the manager and once the workers are ready is given information about the host and port numbers of all workers. Then the process connects and communicates with each worker directly. Control connections are still maintained between the manager and most other processes. They are in particular used for a clean disconnection and shutdown of the whole system.

#### 5.2 Crawling and Indexing Web Pages

The initial step is to get a set of the web pages and organize all the data into a form ready for later usage.

Subsystem	Code Lines	Code Bytes	Logical Modules
Basic Libraries	51,790	1,635,356	49
Web Libraries	15,286	476,084	24
Info. Services	3,950	107,260	13
Data Storage	16,924	566,721	22
Search Engine	79,322	2,855,390	49
<b>Total</b>	<b>167,272</b>	<b>5,640,811</b>	<b>157</b>

Table 1: Yuntis subsystem code size breakdown.

### 5.2.1 Acquiring Initial Data

The data parsing process can read a file with a list of seed URLs or the files that contain the XML dump of the directory structure for the Open Directory Project [27] (publicly available online). These files are parsed while read and appropriate actions are initiated on the database workers to inject this data into the system.

Another way of data acquisition is to parse data from the files for a few essential data tables available from another run of the prototype and rebuild all other data in the system. These essential tables are the log of domain name resolution results for all encountered host names, the log of all URL fetching errors, and the data tables containing compressed raw web pages and robots.txt files [32]. The rebuilding for each of these tables is done by one parser on each cluster workstation that reads and injects into the workers the portion of the data table stored on its cluster node. We use this rebuilding, for example, to avoid refetching a large set of web pages after we have modified the structure of some data tables in the system.

### 5.2.2 Fetching Documents from the Web

The first component of crawling is actual fetching of web pages from web servers. This is done by fetcher processes at each cluster workstation. There is a fetch queue data table that can be constructed incrementally to include all newly encountered URLs. Each fetcher reads from a portion of this queue located on its node and attempts to keep retrieving at most 80 documents in parallel while obeying the robots exclusion conventions [32]. The latter involves retrieving, update, and consulting contents of the robots.txt files for appropriate hosts. To mask response delays of individual web servers, we wish to fetch many documents in parallel, but too many simultaneous TCP connections from our cluster might muscle out other university traffic. If a document is retrieved successfully, it is compressed and stored in the document database partition local to the cluster node, then an appropriate record is added to the document parsing queue. That is, the URL space and the responsibility for fetching and storing it is split among

the cluster nodes, also the split is performed in such a way that all URLs from the same host are assigned to the same cluster node. As a result, in particular to be polite to web servers, the fetcher processes do not need to perform any negotiation with each other and have to communicate solely with local worker processes. The potential downside of this approach is that URLs might get distributed among cluster nodes unevenly. In practice, we saw only 12% deviation from the average of the number of URLs in a node.

### 5.2.3 Parsing Documents

Document parsing was factored into a separate activity because fetching documents is not the only way of obtaining them. Parsing is performed by the parser processes on the cluster nodes. Parsers dequeue information records from the parse queue, retrieve and decompress the documents, and then parse them and inject the results of parsing into appropriate database workers. Parsers start with the portion of the parse queue (and documents) local to their cluster node, but switch to documents from other nodes when the local queue gets empty. Most of the activities in a parser actually happen in a streaming mode: a parser can communicate to the workers some results of parsing the beginning of a long document, while still reading in the end of the document. We also attempt to initiate parsing of at most 35 documents in parallel on each node so that parsers do not have to wait on document data and remote responses from other workers. A planned optimization is to eliminate the cost of page decompression when parsing recently-fetched pages. Another optimization is to have a small dictionary of the most frequent words in each parser so that for a substantial portion of word indexing we can map word strings to internal identifiers directly in the parsers. Having a full dictionary is not feasible as, for instance, we have collected over 60M words for a 10M pages crawl.

### 5.2.4 Word and Link Indexing

We currently parse HTML and text documents. Full text of the documents is indexed along with information about prominence of and distances between words that is derived from the HTML and sentence structure. All links are indexed along with the text that is contained in the link anchor, surrounds the link anchor within a small distance but does not belong to other anchors, and the text of two structurally preceding HTML headers. All links from a web page are also weighted by an estimate of their prominence on the page.

As a result of parsing, we batch the data needed to perform actual indexing into appropriate queues on appropriate worker nodes. After all parsing is done (and during parsing when parsing many documents) these indexing batches are executed according to our general data



manipulation approach. As a result the following gets constructed: unsorted inverted indexes for all words, forward and backward linkage information, and information records about all newly encountered hosts, URLs, sites, and words.

### 5.2.5 Quality-Focused Crawling

Breadth-first crawling can be performed using already described components because the link-indexing process already includes an efficient way of collecting all newly encountered URLs for later fetching. The problem with breadth-first crawling is that it is very likely to fall into (unintended) crawler traps, that is, fetch a lot of URLs few people are going to be interested in. For example, even when crawling the `sunysb.edu` domain of our university a crawler can encounter huge documentation or code revision archive mirrors, many pages of which are quickly reachable with breadth-first search.

We thus employ the approach of crawling focused by page quality scores [7]. In our implementation, after fetching a significant new portion of URLs from an existing fetch queue (for instance, a third of the number of URLs fetched earlier), we execute all operations needed to index links from all parsed documents and then compute page quality scores via our voting model [20] for all URLs and the currently known web linkage graph (see Section 5.3.1). After this the fetch queue is rebuilt (and organized into a priority queue) by filling it with yet unfetched URLs with best scores. Then document fetching and parsing continues with the new fetch queue starting with the best URLs according to the score estimates. Thus, we can avoid wasting resources on unimportant pages.

### 5.2.6 Post-Crawling Data Preprocessing

After we have fetched and parsed some desired set of pages, all indexing operations initiated by parsing must be performed so that we can proceed with further data processing. This involves building lists of backlinks, lists of URLs and sites located on a given host, and URLs belonging to a given site. A site is a collection of web pages assumed to be controlled by a single entity, a person or an organization. Presently any URL is assigned to exactly one site and we treat as sites whole hosts, home pages following the traditional `/~username/` syntax, and home pages located on the few most popular web hosting services. We also merge all hosts with the same domain suffix into one site when they are likely to be under control of a single commercial entity. In order to perform phrase extraction and indexing described later, direct page and link word indexes are also constructed. They are associated with URLs and contain word identifiers for all document words and words associated with links from the document.

## 5.3 Global Data Processing

After collecting a desired set of web pages, we perform several data processing steps that each work with all collected data of a specific kind.

### 5.3.1 Computing Page Quality Scores

We use a version of the developed voting model [20] to compute global query-independent quality scores for all known web pages. This model subsumes Google's PageRank approach [19] and provides us with a way to assess importance of a web page and reliability of the information presented on it in terms of different information retrieval tasks. For example, page scores are used to weigh word occurrence counts, so that unimportant pages can not skew the word frequency distribution. Our model uses the notion of a web site for determining the initial power to influence final scores and to properly discount the ability of intra-site links to increase site's scores. As a result a site can not receive a high score just by the virtue of being large or heavily interlinked, which was the case for the public formulation of PageRank [5, 28].

The score computation proceeds in several stages. The main stage is composed of several iterations, each of which consists of propagating score increments over links and collecting them at the destination pages. As with all other large volume data processing, these score computation steps are organized using our efficient batched execution method. Since in our approach later iterations monotonically incur smaller amount of increments to propagate, the number of increments and the number of web pages that are concerned also reduces. To exploit this we rely more on caching and touching only the needed data at the later iterations.

### 5.3.2 Collecting Statistics

Statistics collection for various data tables happens as a separate step and in parallel on all cluster nodes by sequential scanning of relevant data table partitions, then statistics for different data partitions are joined. The most interesting use of statistics is for estimation of the number of objects that have a certain parameter greater (or lower) than a given value. This is achieved by closely approximating such dependencies using the distribution of the values of such object parameters and the histogram approximations of these distributions. Most types of parameters we are interested in, such as the quality scores for web pages, more or less follow the power-law distribution [1], meaning that most objects have very small values of the parameter that are close to each other and few objects have very high values of the parameter. This knowledge is used when fitting distributions to their approximations and moving the internal boundaries of the collected histograms. As a result in

three passes over data we can arrive at a close approximation that does not significantly improve with more passes.

### 5.3.3 Extracting and Indexing Phrases

To experiment with phrases for instance as keywords of documents we perform phrase extraction and index all extracted phrases. Phrases are simply sequences of words that occur frequently enough. Phrase extraction is done in several stages corresponding to the possible phrase lengths (presently we limit the length to four). Each stage starts with considering forward word/phrase indexes for all documents that constitute top 20% with respect to page quality scores from Section 5.3.1. We thus both reduce processing costs and can not be manipulated by low-score documents. All sequences of two words (or a word and a phrase of length  $i-1$ ) are counted (weighted by document quality scores) as phrase candidates. The candidates with high scores are chosen to become phrases. New phrases are then indexed by checking for all “two-word” sequences in all documents if they are really instance of chosen phrases. Eventually complete forward and inverted phrase indexes are built.

This phrase extraction algorithm is purely statistical: it does not rely on any linguistic knowledge. Yet it extracts many common noun phrases such as “computer science” or “new york”, although along with incomplete phrases that involve prepositions and pronouns like “at my”. Additional linguistic rules can be easily added.

### 5.3.4 Filling Page Scores into Indexes

In order to be able to answer user queries quicker, it is beneficial to put page quality scores for all URLs into all inverted indexes and sort the lists of URLs in them by these scores. Consecutively, to retrieve a portion of the intersection or the union of URL lists for several words with highest URL scores (which constitutes the result of a query), we do not have to examine the whole lists.

Page quality scores are filled into the indexes simply by consulting the score values for the appropriate URLs in the URL information records, but this is done via our delayed batched execution framework so that the information records for all URLs do not have to fit into the memory of the cluster. To save space we map four-byte floating point score values to two-byte integers using a mapping derived from approximating the distribution of score values (see Section 5.3.2).

In addition we also fill similar approximated scores into indexes of all words (and phrases) associated with links pointing to URLs. To do this we also keep full URL identifiers of linking URLs in these indexes. This allows us to quickly assess the weight of a all words used to describe a given URL via incoming links. Sorting of

various indexes by the approximated page score values is done as a separate data table modification step.

### 5.3.5 Building Linkage Information

We use a separate stage to construct the data tables about backward URL to URL linkage, forward web site to URL on other sites linkage, and backward URL on other sites to site linkage from forward URL to URL linkage data. At this time we also incorporate page quality scores into these link lists for later use during querying.

### 5.3.6 Extracting Keywords

We compute characteristic keywords for all encountered pages to be later used for assessing document similarity. They are also aggregated into keyword lists for web sites and ODP [27] categories. All these keyword lists are later served to the user as a part of the information record about an URL, site, or category.

Keywords are words (and phrases) most related to a URL and are constructed as follows: inverted indexes for all words that are not too frequent and not too rare (as determined by hand-tuned bounds) are examined and candidate keywords are attached to the URLs that would receive the highest scores for a query composed of the particular word. For all word-URL pairs, the common score boundary to pass as a candidate keyword is tuned to reduce processing, while yielding enough candidates to choose from. Since both document and link text indexes are considered similarly to query answering, extracted document keywords are heavily influenced by the link text descriptions. Thus, we are sometimes able to get sensible keywords for not fetched documents.

For all URLs the best of candidate keywords are chosen according to the following rules: we do not keep more than 30 to 45 keywords per URL depending on URL's quality score, and we try to discard keywords that have scores smaller by a fixed factor on the log-scale than the best keyword for the URL. We also discard keywords that have a phrase containing them as another candidate keyword of the same URL with a score of the same magnitude on log-scale. The resulting URL keywords are then aggregated into candidate keyword sets for sites and categories, which are then similarly pruned.

### 5.3.7 Building Directory Data

The Open Directory Project [27] freely provides a classification directory structure similar to Yahoo [39] in size and quality. Any selected part of the ODP's directory structure can be imported and incorporated by Yuntis. Description texts and titles of listed URL's are fully indexed as a special kind of link texts. All the directory structure is fully indexed, so that the user can easily navigate and search the directory or its parts, as well as see in what categories a given URL or a subcategory are men-



tioned. For some reason the latter feature appears to be unique to Yuntis despite the numerous web sites using ODP data.

One interesting problem we had to resolve was to determine the portion of subcategory relations in the directory graph that can be treated as subset inclusion relations. Ideally we want to treat all subcategory relations this way, but this is impossible since the subcategory graph of ODP is not acyclic in general. We ended up with an efficient iterative heuristic graph-marking algorithm, that likely can be improved, but behaves better in the cases we tried than corresponding methods in ODP itself or Google [11]. Note that all the directory indexing and manipulation algorithms are distributed over the cluster and utilize the delayed batched execution framework.

### 5.3.8 Finding Similar Pages

Locating web pages that are very similar in topic, service, or purpose to a given (set of) pages (or sites) is a very useful web navigation tool on its own. Algorithms and techniques used for finding similar pages can also be used for such tasks as clustering or classifying web pages.

Yuntis precomputes lists of similar pages for all pages (and sites) with respect to three different criteria: pages that are linked from high-score pages closely with the source page, pages that have many high-scored keywords common with the source page, and pages that link to many of the pages the source page does. The computation is organized around efficient volume processing of all relevant pieces of "similarity evidence". For example, for textual similarity we go from pages to all their keywords, find other pages that have the same word as a keyword, choose the highest of these similarity evidences for each word, and send them to the relevant destination pages. At each destination page all similarity evidences from different keywords for the same source page are combined and some portion of most similar pages is kept. As a result, all processing consumes linear time in the number of known web pages, and the exact amount of processing (and similar pages kept) can be regulated by the values that determine what evidence is good enough to consider (or to qualify for storage).

### 5.4 Data Compaction and Checkpointing

Data table partitions organized as heaps of variable-sized records usually have many unused gaps in them after being extensively manipulated: Empty space is reserved for fast expected future growth of records. Not all space freed after a record shrinking is later used for another record. To reclaim all this space on disk we introduced a data table compaction stage that removes all the gaps in heap-like table partitions and adjusts all indexes to the

records in each such partition that are contained in the associated information partition. The latter is cheap to accomplish as each such information partition easily fits in memory.

All data preparation in Yuntis is organized in stages that roughly correspond to the previous subsections. To alleviate the consequences of hardware and software failures, we introduced data checkpointing after all stages that take considerable time, as well as the option to restart processing from any such checkpoint. Checkpointing is done by synchronizing all data from memory to files on disks and "duplicating" the files by making new hard links. When we later want to modify a file with more than one hard link, we first duplicate its data to preserve the integrity of earlier checkpoints.

### 5.5 Answering User Queries

User queries are answered by any of the web server processes; they handle HTTP requests, interact with database workers to get the needed data, and then format the results into HTML pages. Query answering for standard queries is organized around sequential reading, intersecting and merging of the beginnings of relevant sorted inverted indexes according to the structure of the query. Then additional information for all candidate and resulting URLs is queried in parallel, so that URLs can be clustered by web sites and URL names and document fragments relevant to the query can be displayed to the user. For flexible data examination, Yuntis supports 13 boolean-like connectives (such as OR, NEAR, ANDNOT, and THEN) and 19 types of basic queries that can be freely combined by the connectives. In many cases exact information about intra-document positions is maintained in the indexes and utilized by connectives. An interesting consequence of phrase extraction and indexing it that it can considerably speed up (for example, by a factor of 100) many common queries that (implicitly) include some indexed phrases. In such cases, the work of merging the indexes for the words that form the phrase(s) has been already done during phrase indexing.

## 6 Performance Evaluation

Below we describe the hardware configuration of our cluster and discuss the measured performance and sizes of the handled datasets.

### 6.1 Hardware Configuration

Presently Yuntis is installed on a 12-node cluster of Linux PC workstations each running a Red Hat Linux 8.0 with a 2.4.19 kernel. Each system has one AMD Athlon XP 2000 CPU with 512MB of DDR RAM connected by a 2\*133MHz bus, as well as two 80GB 7200 RPM Maxtor EIDE disks model 6Y080L0 with average seek time of 9.4msec. Two large partitions on each

Documents Stored	4,065,923
URLs Seen	34,638,326
Hyper Links Seen	87,537,723
Inter-Site Links	18,749,662
Web Sites Recognized	2,833,110
Host Names Seen	3,139,435
Canonical Hosts Seen	2,448,607
Words Seen	30,311,538
Phrases Extracted	574,749
Avg. Words per Document	499.5
Avg. Links per Document	20.4
Avg. Document Size	13.1 KB
Avg. URLs per Site	12.2
Avg. Word Length	9.89 char-s
Total Final Data Size	87,446 MB
Avg. Data per Document	21,039 B
Compressed Documents	13,739 MB
Inverted Doc. Text Indexes	23,188 MB
Inverted Link Text Indexes	11,125 MB
Other Word Data	1,628 MB
Keyword Data	1,922 MB
Page Similarity Data	25,908 MB
All Linkage Indexes	2,861 MB
Other URL Data	4,613 MB
Other Host, Site, and Category Data	2,458 MB
Forward Word Indexes (not in total)	29,528 MB

Table 2: Data size statistics.

two disks are joined by LVM [22] into one 150G ext3 file system for data storage. The nodes are connected into a local network by full-duplex 100Mbps 24-port CompeX SRX 2224 switch and 12 network cards. The ample 4.8Gbps back plane capacity of the switch ensured that it will not become the bottleneck of our configuration. The full-duplex 100Mbps connectivity of each cluster node has not yet become a performance bottleneck: So far we have seen sustained traffic of around 7MBps in and 7MBps out for each node out of the potentially available 12.5+12.5MBps. With additional optimizations or increase of CPU power leading to higher communication volume generated by each node, we might have to use a higher-capacity cluster connectivity, for instance, channel bonding with several 100Mbps cards per node. A central management workstation with an NFS server is also connected to the switch, but does not noticeably participate in the workloads of Yuntis, simply providing a central place for logs, configuration files, and some-

times the executables. The cluster is connected to the outside world via the 100Mbps campus network and a 155Mbps OC3 university link.

## 6.2 Dataset Sizes

Table 2 provides various size statistics. We can for example see that the bulk of the stored data falls on the text indexes, similarity lists, and compressed documents. This data and later performance figures are for a particular crawl of 4 million pages started by fetching 1.3 million URLs listed in the non-international portion of ODP. All these numbers simply provide order-of-magnitude estimates of the typical figures one would get for similar datasets on similar hardware.

We use the bzip2 library [6] for compressing individual web pages. This achieves a compression factor of 3.87. bzip2 is very effective when applied to individual pages: compressing the whole document data table partitions would save only 0.38% of space. We have not yet seriously considered additional compression of other data tables beyond compact data representation with bit fields.

## 6.3 Data Preparation Performance

As we have demonstrated [21], the batched data-driven approach to data manipulation leads to performance improvements by a factor of 100 via both better CPU and memory file cache utilization. It also makes the performance much less sensitive to the amounts of available memory.

Table 3 provides various performance and utilization figures for different stages of data preprocessing. The second to last column gives the amount of file data duplication needed to maintain the checkpoint for the previous stage. The data shows that phrase extraction, similarity precomputation, and filling of scores into indexes are the most expensive tasks after the basic tasks of parsing and indexing. Various stages exhibit different intensity of disk and network I/O according to their nature, as well as perform differently in terms of CPU utilization. We are planning to instrument the prototype to determine, for each stage, the exact contributions of the possible factors to non-100% CPU utilization, and then improve on the discovered limitations.

Peak overall fetching speed reaches 217 doc-s/sec and 3.6 MB/sec of document data. Peak parsing speed reaches 935 doc-s/sec. Sustained speed of parsing with complete indexing of encountered words and links is 304 doc-s/sec. Observed overall crawling speed when crawling 4M pages with fetch queue rebuilding after getting each 0.6M pages was 67 doc-s/sec. The speed to do all parsing and complete all incurred indexing is 297 doc-s/sec. The speed of all subsequent preprocessing is 90 doc-s/sec. The total data preparation

Processing Stage	Time (min)	CPU Utilization (%)			Disk I/O (MB/s)		Netw. I/O (MB/s)		Total Disk Data (GB)	
		User	Sys.	Idle	In	Out	In	Out	Cloned	Kept
Doc. Parsing & Indexing	168	55	7	35	1.2	1.9	1.2	1.1	9	73
Post-Parsing Indexing	69	46	9	42	1.8	2.6	1.2	1.2	54	76
Pre-Score Statistics	6	43	3	46	14.8	0.03	0.01	0	0	76
Page Quality Scores	69	27	13	57	1.6	6.4	0.77	0.77	3	76
Linkage Indexing	9	63	10	24	1.9	3.2	1.8	1.8	4	76
Phrase Extr. & Indexing	276	39	12	47	2.9	2.2	2.3	2.3	52	110
Word Index Merging	25	28	11	58	0.57	3.4	0	0	53	110
Scores into Word Index	87	57	16	25	2.3	2.5	3.9	3.9	53	110
Scores into Link Text Index	42	51	12	34	1.9	1.8	3.0	2.9	20	110
Word Statistics	33	57	2	37	6.8	0.09	0.30	0	1	110
Choosing Keywords	44	29	7	59	3.0	1.8	0.66	0.60	53	113
Building Directory Data	8	26	5	66	1.7	2.8	0.67	0.64	2	117
Word Index Sorting	23	28	8	60	0.4	3.4	0.02	0	54	117
Finding Similar Pages	116	44	11	43	2.7	2.3	2.5	2.5	0	143
Scores into Other Indexes	33	63	19	17	2.2	2.3	4.4	4.4	12	143
Sorting Other Indexes	5	33	2	54	0.05	4.8	0.09	0	14	143
Data Table Compaction	13	15	12	66	10.0	7.7	0	0	4	117

Table 3: Average per-cluster-node data processing performance and resource utilization.

speed excluding fetching of documents it thus 69 doc/sec. Provided this performance scales perfectly with respect to both the number of cluster nodes and the data size, it would take a cluster of 430 machines to process in two weeks the  $3 \cdot 10^9$  pages presently covered by Google [11], which looks like a quite reasonable requirement.

## 6.4 Query Answering Performance

Because Yuntis was built for experimenting with different search engine data preprocessing stages, we did not optimize query answering speeds beyond a basic acceptable level. Single one-word queries usually take 10 to 30sec when no relevant data is in memory and 0.1 to 0.5sec when all needed data is in the memory of the cluster nodes. The longer times are dominated by the need to consult a few thousands of URL information records scattered on the disks. We currently do not have any caching of (intermediate) query results, except the automatic local file data caching in memory by the OS. The performance for multiword queries heavily depends on the specific words used: our straightforward sequential intersecting of word indexes would be significantly outperformed by a more optimized zig-zag merging based on binary search in the cases when very large indexes yield a much smaller intersection.

## 6.5 Quality of Search Results

To illustrate the usability of Yuntis we provide the samples in Table 4, report that Yuntis served 125 searches per day on average in February 2003, and encourage the reader to try it for themselves at <http://yuntis.ecsl.cs.sunysb.edu/>.

## 7 Enhancements and Future Directions

There are a number of general Yuntis improvements and extensions one can work on such as overall performance and resource utilization optimizations (especially for query answering), better tuning of various parameters, and implementation of novel searching services, for instance, classifying pages into ODP's directory structure [27]. As Table 3 shows, phrase extraction and indexing is one of the most important areas of performance optimization. One approach is to move phrase indexing into the initial document parsing and derive the phrase dictionary in a separate stage using a much smaller subset of good representative documents.

Another significant project is to build support for automatic work rebalancing and fault tolerance, so that cluster nodes can automatically share the work most equally, be seamlessly added, or go down during execution affecting only the overall performance. The approach here can be to consider sets of related partitions together with different batches of operations to them as the atomic units of data and work to be moved around.



Results for query	Keywords for	Pages linked like	Pages textually like
university	www.apple.com	www.subaru.com	www.cs.sunysb.edu
www.indiana.edu	apple computer inc	www.toyota.com	www.sunysb.edu
www.umich.edu	apple macintosh	www.vw.com	www.cs.uiuc.edu
www.stanford.edu	apple computers	www.saabusa.com	www.cs.umass.edu
www.wsu.edu	macintosh computer	www.pontiac.com	www.cs.berkeley.edu
www.uiuc.edu	macintosh computers	www.suzuki.com	www.cs.colorado.edu
www.cam.ac.uk	apple and	www.porsche.com	www.cs.man.ac.uk
www.about.bham.ac.uk	quick time	www.oldsmobile.com	www-cs.stanford.edu
www.cmu.edu	apple has	www.saturncars.com	www.cs.virginia.edu
www.msu.edu	computer the	www.volvocars.com	www.cs.unc.edu
www.cornell.edu	made with macintosh	www.mazdausa.com	www.suny.edu

Table 4: Top ten results for four typical Yuntis queries.

An important scalability (and work balancing) issue is posed by the abundance of power-law distributed properties [1] on the Web. As a result, many data tables have few records that are *very* large, for example, individual occurrence lists for most frequent words grow over 1GB at around 10 million document datasets, whereas most other records in the same table are under few KB. Handling (reading, updating, appending, or sorting) such large records efficiently requires special care such as not attempting to load (or even map) them into memory as a whole, and working with them sequentially or in small portions. In addition, such records reflect poorly on the ability to divide the work equally among cluster nodes by random partitioning of the set of records. A known solution is to split the records into subparts; for example, a word occurrence list can be divided according to a partitioning of the whole URL space. We are planning to investigate the compatibility of this approach with processing tasks that need to consider such records as a whole, and whether it is best to do this splitting for all records in a table or only for the extremely large ones.

## 8 Conclusions

We have described the software architecture, major employed abstractions and techniques, and implementation of the main processing tasks of Yuntis, a 167,000 lines feature-rich operational search engine prototype. We have also discussed its current configuration, its performance, and the characteristics of handled datasets, as well as outlined some existing problems and roads for future improvements.

The implementation of Yuntis allowed us to experiment with, evaluate, and identify several enhancements of our voting model [20] for assessing quality and relevance of web pages. The same is true for other search engine functions (such as phrase indexing, keyword extraction, similarity lists precomputation, and directory data usage), as well as their integration in one system —

all while working with realistic datasets of millions of web pages.

The most important contributors to this success were the following: First, the approach of data partitioning and operation batching provided high cluster performance without task-specific optimizations leading to convenience of implementation and faster prototyping. Second, the modular, layered, and typed architecture for data management and cluster-based processing allowed us to build, debug, extend, and optimize the prototype rapidly. Third, the event-driven call/callback processing model was useful for allowing us to have a relatively simple, efficient, and coherent design of all components of our comprehensive search engine cluster.

## Acknowledgments

This work was supported in part by NSF grants IRI-9711635, MIP-9710622, EIA-9818342, ANI-9814934, and ACI-9907485. The paper has greatly benefited from the feedback of its shepherd, Erez Zadok, and the USENIX anonymous reviewers.

## Availability

The Yuntis prototype can be accessed online at <http://yuntis.ecsl.cs.sunysb.edu/>. Its source code is available for download at <http://www.ecsl.cs.sunysb.edu/~maxim/yuntis/>.

## References

- [1] Lada A. Adamic. Zipf, power-laws, and pareto - a ranking tutorial. Technical report, Xerox Palo Alto Research Center, 2000.
- [2] The Apache Web Server, [www.apache.org](http://www.apache.org).
- [3] The Berkeley Database, [www.sleepycat.com](http://www.sleepycat.com).
- [4] Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian. The Connectivity Server: fast access to linkage information on the Web. In *Proceedings*

- of 7th International World Wide Web Conference, 14–18 April 1998.
- [5] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of 7th International World Wide Web Conference*, 14–18 April 1998.
  - [6] The bzip2 Data Compressor, [www.digistar.com/bzip2](http://www.digistar.com/bzip2).
  - [7] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through URL ordering. In *Proceedings of the Seventh World-Wide Web Conference*, 1998.
  - [8] The Common Object Request Broker Architecture, [www.corba.org](http://www.corba.org).
  - [9] The Distributed Component Object Model, [www.microsoft.com/com/tech/DCOM.asp](http://www.microsoft.com/com/tech/DCOM.asp).
  - [10] The GNU Project Debugger, [sources.redhat.com/gdb](http://sources.redhat.com/gdb).
  - [11] Google Inc., [www.google.com](http://www.google.com).
  - [12] Allan Heydon and Marc Najork. Mercator: A scalable, extensible Web crawler. *World Wide Web*, 2(4):219–229, December 1999.
  - [13] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. WebBase: A repository of web pages. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, Netherlands, May 2000.
  - [14] The ht://Dig Search Engine, [www.htdig.org](http://www.htdig.org).
  - [15] The Isearch Text Search Engine, [www.cnidr.org/isearch.html](http://www.cnidr.org/isearch.html).
  - [16] Dan Kegel. The C10K Problem, [www.kegel.com/c10k.html](http://www.kegel.com/c10k.html).
  - [17] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 668–677, San Francisco, California, 25–27 January 1998.
  - [18] Jonathan Lemon. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track (USENIX-01)*, pages 141–154, Berkeley, California, June 2001.
  - [19] Maxim Lifantsev. Rank computation methods for Web documents. Technical Report TR-76, ECSL, Department of Computer Science, SUNY at Stony Brook, Stony Brook, New York, November 1999.
  - [20] Maxim Lifantsev. Voting model for ranking Web pages. In Peter Graham and Muthucumaru Maheswaran, editors, *Proceedings of the International Conference on Internet Computing*, pages 143–148, Las Vegas, Nevada, June 2000.
  - [21] Maxim Lifantsev and Tzi-cker Chiueh. I/O-conscious data preparation for large-scale web search engines. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases*, August 20–23, 2002, Hong Kong, China.
  - [22] The Logical Volume Manager, [www.sistina.com/products\\_lvm.htm](http://www.sistina.com/products_lvm.htm).
  - [23] The Linux Virtual Server, [www.linuxvirtualserver.org](http://www.linuxvirtualserver.org).
  - [24] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the Web. In *Proceedings of the 10th International World Wide Web Conference*, Hong Kong, May 2001.
  - [25] The Message Passing Interface, [www-unix.mcs.anl.gov/mpi](http://www-unix.mcs.anl.gov/mpi).
  - [26] The GNU Nana Library, [www.gnu.org/software/nana](http://www.gnu.org/software/nana).
  - [27] The Open Directory Project, [www.dmoz.org](http://www.dmoz.org).
  - [28] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the Web. Technical report, Stanford University, California, 1998.
  - [29] Jef Poskanzer. Web Server Comparisons, [www.acme.com/software/tthttpd/benchmarks.html](http://www.acme.com/software/tthttpd/benchmarks.html).
  - [30] The Parallel Virtual Machine, [www.csm.ornl.gov/pvm](http://www.csm.ornl.gov/pvm).
  - [31] Berthier Ribeiro-Neto, Edleno S. Moura, Marden S. Neubert, and Nivio Ziviani. Efficient distributed algorithms to build inverted files. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Information Retrieval*, pages 105–112, Berkeley, California, August 1999.
  - [32] Web Robots Exclusion, [www.robotstxt.org/wc/exclusion.html](http://www.robotstxt.org/wc/exclusion.html).
  - [33] The Simple Object Access Protocol, [www.w3.org/TR/SOAP](http://www.w3.org/TR/SOAP).
  - [34] The Standard Template Library, [www.sgi.com/tech/stl](http://www.sgi.com/tech/stl).
  - [35] The Simple Web Indexing System for Humans, [swish-e.org](http://swish-e.org).
  - [36] The tthttpd Web Server, [www.acme.com/software/tthttpd](http://www.acme.com/software/tthttpd).
  - [37] The Webglimpse Search Engine Software, [webglimpse.net](http://webglimpse.net).
  - [38] The eXternalization Template Library, [xtl.sourceforge.net](http://xtl.sourceforge.net).
  - [39] Yahoo! Inc., [www.yahoo.com](http://www.yahoo.com).

# CSE — A C++ Servlet Environment for High-Performance Web Applications

Thomas Gschwind      Benjamin A. Schmit

*Abteilung für Verteilte Systeme*

*Technische Universität Wien*

*Argentinierstraße 8/E1841*

*A-1040 Wien, Austria, Europe*

{tom,benjamin}@infosys.tuwien.ac.at

<http://www.infosys.tuwien.ac.at/Staff/{tom,benjamin}/>

## Abstract

Current environments for web application development focus on Java or scripting languages. Developers that want to or have to use C or C++ are left behind with little options. We have developed a C/C++ Servlet Environment (CSE) that provides a high performance servlet engine for C and C++. One of the biggest challenges we have faced while developing this environment was to come up with an architecture that provides high performance while not allowing a single servlet to crash the whole servlet environment, a serious risk with C and C++ application development. In this paper we explain our architecture, the challenges and trade-offs we have faced, and compare the performance of our environment to that of top servlet environments available today.

## 1 Introduction

Every day, the web is applied to more and new application domains. In some cases people try to convert services that historically have been handled by a different mechanism, such as USENET News, to the web. In other cases, such as email, they are complemented by a mechanism that allows users to access these services via a web front-end. Due to this success and the increasing number of application domains web server performance is of major importance [8, 11] and unresponsive and slow web sites may send users seeking for alternatives [5].

Traditionally, web services have been implemented using CGI programs in the form of compiled C and C++ programs or Perl scripts that were executed by the web server. Although these approaches work perfectly fine for simple dynamic web content they are cumbersome to use if a whole business process should be modeled as a web application. This stems from the fact that they do not take care of the state-management between subsequent web requests. These issues, however, are taken care of by newer approaches such as the Java Servlet

Technology [22], the Python Zope Server [14], or dedicated application servers (e.g., Bea Weblogic or JBoss).

One advantage of servlets is that they are executed as part of a servlet environment that, unlike CGI scripts, needs not be restarted at each invocation. To protect one servlet from another, these environments use programming languages that take care of memory management issues. Another advantage is that these languages come bundled with standardized libraries providing support for numerous different tasks.

These advantages, however, have a price. Legacy applications that make use of C or C++ cannot be easily integrated. Although technologies such as SWIG [4] or JNI [16, 25] simplify the integration of legacy applications into scripting languages they still require developers of such systems to deal with different systems. Sometimes the choice of language is mandated by management or the developers simply prefer the use of C or C++.

Another advantage of C and C++ is that these languages provide better performance and a finer grained integration of the operating system's security mechanisms. This is probably one of the reasons why the Apache HTTP Server and the Microsoft Internet Information Server, the two most prominent web servers with a combined market share of over 85% [20], are written in C and C++ respectively.

In this paper, we present a servlet environment that is completely implemented in C++. To the best of our knowledge, our C/C++ Servlet Environment (CSE) is the first servlet environment that uses the potential of C++. Similar to the servlet engines implemented in Java, the architecture of our servlet environment offers adequate protection between the individual servlets to be executed. Hence, our servlet engine provides the following advantages over those using Java or scripting languages:

- It supports the integration of existing C/C++ code into web applications without mixing different programming languages and converting data types.



- It provides a better integration into the security mechanisms provided by the operating system.
- The C++ programming language allows developers to write more efficient code since they can choose from a wider range of implementation choices [24].

This paper is structured as follows. In Section 2 we present the requirements for a servlet environment as well as the advantages and challenges of using C++. Section 3 shows how these challenges have influenced the design and implementation of the C/C++ Servlet Environment, and application development for the CSE is discussed in Section 4. Related work is presented in Section 5 and in Section 6 we compare the performance of our approach with that of other approaches. Future work is discussed in Section 7 and we draw our conclusions in Section 8.

## 2 Requirements

The main goals during the design of the C/C++ Servlet Environment were security, stability, ease of use, parallelism, and performance. Before we can have a closer look at these requirements, however, it is necessary to give a brief overview of the typical architecture of a web site. This architecture is depicted in Figure 1.

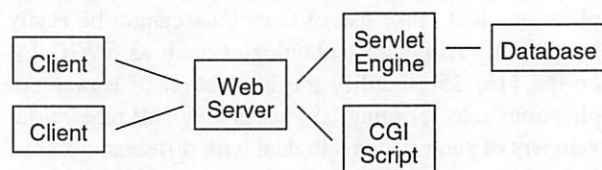


Figure 1: A Typical Web Site Implementation

A web site must include a web server that handles requests from multiple clients. Requests that cannot be handled by the web server itself are forwarded to a CGI program or a servlet engine where a web service is executed. The web service in turn may contact a database or application server for persistent data storage. Since HTTP is stateless [7], the client has to transmit the application's current state or a session identifier with each request. In the latter case a mapping from session identifier to state has to be maintained by the servlet engine.

CGI programs have the drawback that they have to be executed anew for each client's request. Hence, they need to be started at each request and then have to read in their configuration data and session state from persistent storage. A servlet engine, on the other hand, is running all the time and keeps several servlets as well as their session state in memory. Servlets need to maintain the client's session state (for instance, the contents of a shopping basket) because web clients only provide limited functionality to maintain this kind of data.

Since many servlet engines execute several different servlets within the same process *stability* is a major concern. If one servlet crashes, the other servlets have to continue to run unaffectedly. For a servlet engine, it is very important to recognize and handle this kind of failure since there is no way of judging the stability of a user-supplied servlet from within the servlet engine.

Stability is one reason why Java and interpreted languages in general are popular for this task. If a Java servlet crashes only its thread of execution is terminated and all the other servlets continue to run. Hence, the worst that can happen is that a servlet consumes overly much resources such as processor time, memory in the form of unused but still referenced Java objects, or network bandwidth. The price for these benefits is that all servlets are executed with the same privileges and that the operating system's security mechanisms need to be re-implemented as part of the servlet engine. Another drawback is a slight performance overhead since Java does not allow developers to write code on a level as low as it can be written with C and C++.

The disadvantage of C and C++ is that these programming languages are not as safe. Bugs in C and C++ programs typically take down the whole process and they tend to get apparent at a much later time (typically at a point of execution that is unrelated to the place that caused the error). Therefore, even if the application is catching the signal that a segmentation violation has occurred, recovery is difficult. Hence, the design of a servlet engine written in these languages has to solve those challenges.

*Security* is necessary to minimize the chance that servlets can be exploited by an intruder. Our architecture reuses the security mechanisms that have been built into the operating system. It allows sandboxes, and thus servlets, within the same servlet environment to be executed with different user privileges. The advantage of this approach is that system administrators can use standard access privilege mechanisms and do not have to get familiar with a new security management system which can lead to misunderstandings. Additionally, our approach requires only a single security mechanism to be checked for possible vulnerabilities.

A disadvantage of C and C++, however, is that such programs are open to buffer overflow attacks if they have not been implemented carefully. The threat and impact of such attacks can be minimized by using the C++ Standard Template Library which has been designed in order to minimize such programming mistakes and by using operating system mechanisms such as a non-executable user stack area. No programming language or servlet environment, however, can guarantee that a program or servlet cannot be exploited by an intruder. The final responsibility is always up to the developer.

*Ease of use* is another important aspect for a servlet engine. Even though there are servlet engines for Java that provide good performance, these servlet engines are of little use to C or C++ programmers that want to use existing code for their web applications. Using C and C++ in combination with such servlet engines requires the use of the Java Native Interface (JNI) [16, 25], the conversion between C and Java data types, and to deal with low-level details of both languages. Although systems such as SWIG [4] can be used to help with this issue they do not solve the problem of having to deal with two different languages. Hence, a C++ servlet environment is more convenient to use for developers that have to deal with C or C++ code.

A good servlet engine must not only be easy to use but has to provide good *performance* as well. This gets apparent by the fact that the two most popular web servers are implemented in C and C++ respectively. A convenient servlet environment that requires load balancing among multiple servers to be able to handle the incoming requests loses much of its original appeal. This is one of the reasons why we have developed the CSE. The CSE was designed towards optimal performance.

*Parallelism* is, on a server machine, a prerequisite for performance and scalability. On a network, the upper bound of the access time to a service can be high and thus forbids handling requests in serial. The CSE introduces parallelism by providing several services on a single machine, by partitioning a service into several parts with simple interfaces between them, and by replicating those parts. Additionally, compared to using interpreted languages that use a global interpreter lock for thread locking [28, Section 8.1], the approach to use C++ has the advantage of having a fine grained thread locking mechanism as provided by linux-threads or pthreads [15].

### 3 Design and Implementation

Figure 2 shows the architecture of our C/C++ Servlet Environment. It consists of an Apache module, a servlet server and several sandbox processes. The CSE uses the Apache server to handle incoming HTTP requests, the servlet server is used for the management of the sandbox processes and the sandboxes are responsible for the execution of the servlets. We use different sandbox processes because this protects a servlet in one sandbox from an unstable one in another sandbox. Additionally, this approach allows administrators to execute different sandboxes with different user privileges. Although we have used C++ for the implementation, our architecture can be easily reused for a servlet environment implemented in plain C.

#### 3.1 The Apache Module

We use the Apache web server [1] to handle HTTP requests. This approach has several advantages over implementing a new web server for our servlet engine such as provided by the Tomcat Servlet Engine [2].

- Apache uses a modular design and can be extended easily.
- It is implemented in C facilitating data exchange with our C++ servlet engine. C structures are compatible to C++ structures.
- Its add-on modules provide features that we would never have implemented as part of CSE.
- It has a 60% market share [20] and has proven to be a reliable web server.
- It is free software.

Another advantage of the modular design we have chosen is that only the web server's *Servlet Module* has to be re-implemented if a different web server has to be used for a given web site.

The Apache Module registers the URLs that are implemented by the servlet engine's servlets and the C++ Server Pages. Requests to other URLs such as static web pages or images are handled by Apache itself without consulting our module. Servlet requests are forwarded to the servlet server which assigns them to one of the sandboxes. One risk of this approach is that the servlet server might become a bottleneck. Hence, in future versions of CSE, we plan to extend the Apache module such that the requests are sent to the appropriate sandbox directly. Going a step further and executing the servlets by the Apache module directly, however, is not possible since that would compromise the web server's stability.

#### 3.2 The Servlet Server

The *Servlet Server* is responsible for the management of the sandboxes. To do that it processes requests forwarded by the Apache module and determines, based on the servlet registry, the sandbox that is responsible for the servlet's execution. The *Servlet Registry* maintains a mapping of the servlets and the sandboxes they should be executed in. This mapping defined within the server's configuration file.

If a C++ Server Page (CSP) is used, the *CSP Manager* converts the CSP document into a servlet and compiles it. Several *Compiler Threads* can be started by the server in order to compile CSPs when they are first requested or when they have changed. They are synchronized so that no more than a single thread for a single servlet can run at a time. The compiled servlet is put into a cache directory, along with the servlet source code, a configuration file containing the destination sandbox, and an error output file which also serves as a timestamp of the last compilation attempt. A CSP is only recompiled if

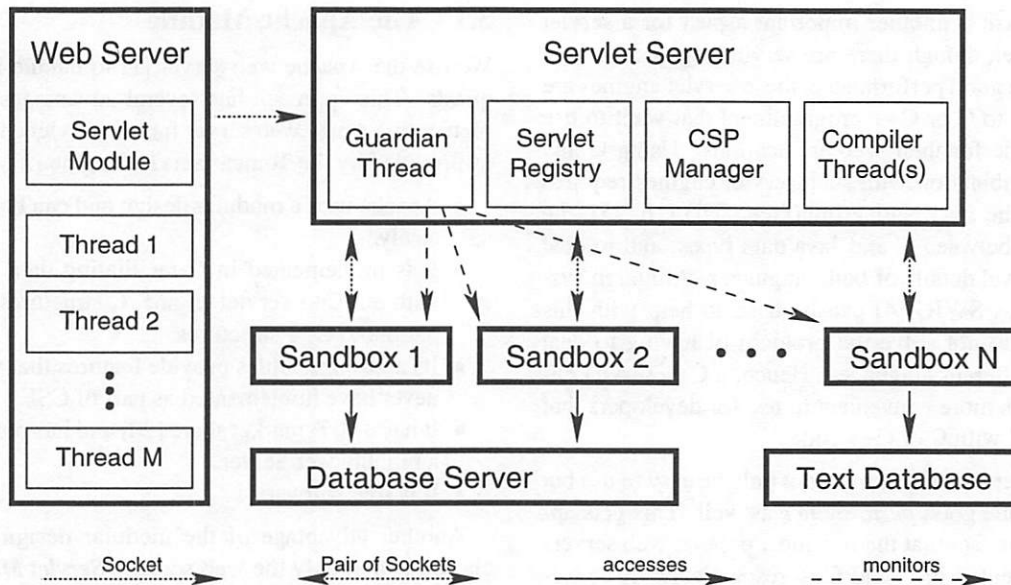


Figure 2: CSE Architecture

its timestamp is newer than the timestamp of its error output file. We do not use the CSP's object file since it is only created if the compilation is successful. If the sandbox does not exist yet, it will be created after compilation. Then, the destination sandbox is configured to host the new servlet. If for some reasons, however, CSP compilation during run-time is undesired or impossible, it is possible to compile them before starting the server.

A *Guardian Thread* watches over the state of the sandboxes and restarts them if an unstable web application crashes. Instead of using a separate thread that sleeps most of the time, it should also be possible on UNIX systems to catch the child signal instead. From the performance point of view, however, this poses no difference.

In order to minimize downtimes of the servlet server, its configuration can be changed dynamically. The configuration file tells the servlet registry which servlets should be loaded into which sandboxes. When the application server processes a request, it first checks the timestamp of the configuration file. If there was a change (or there was no request after startup yet), the configuration file is read, and the internal configuration data is updated. Part of the configuration data, such as the location of the shared object files, is passed on to the sandboxes where it is needed. This update process is also invoked when a sandbox has to be restarted by the guardian thread.

### 3.3 The Sandboxes

Several separate tasks, the *Sandboxes*, handle the actual execution of the servlets. Each sandbox may encapsulate one or more web applications consisting of one or more

servlets. The purpose of using several sandboxes is to take care of the session management and to provide a barrier for unstable servlets.

The ability to execute several servlets within the same sandbox increases the scalability of our servlet environment. This approach allows developers to place servlets that frequently need to interact with each other into the same sandbox and hence enables a more efficient communication between the servlets and reduces the number of context switches required.

The sandbox processes read requests from an input socket and execute them. The servlets themselves are loaded as dynamically shared objects. C++ Server Pages are translated into servlets which in turn are compiled into shared objects. Shared objects can be loaded into a running program on demand by a system call. The disadvantage, however, is that the server's original programmer can never be sure about their stability.

If a servlet crashes while processing a request, the servlet server's guardian thread notices the crash and restarts and reinitializes the failed sandbox. If a request comes in while the destination sandbox is down, it is buffered and executed as soon as the sandbox is up and running again.

This architecture has already proved its usefulness: Our original implementation contained an error that would crash every sandbox after a little more than 1000 requests. We have not discovered this error until we have started running the benchmarks because the crashed sandboxes were always restarted, and (except for a slight performance loss) no problem was visible.

Additionally, since each sandbox is executed within its own process, system administrators may choose to



run different sandboxes with different user privileges. This architecture provides a finer grained access control than Java servlet environments which execute all servlets within the same process and thus with the same user privileges.

### 3.4 Persistent Storage

The ability to provide persistent storage is of major importance for most web applications. A voting application, for example, has to manage the votes cast by its users. The persistence mechanism of the CSE can be used internally by the sandbox's session management if the servlet's `sessionType` attribute is set to `DatabaseSession`. Alternatively, it may be used directly by the application developer.

The C++ Servlet Environment has been designed so that the persistence mechanism can be replaced easily. The persistence mechanism is encapsulated by a traits class [19, 24] with which its classes are parameterized. A traits class can be compared to a set of callback functions with the difference that they are known during compile time and thus leave more room for optimization. This approach allows application developers to choose a persistence level that fits their needs best, e.g., a database such as MySQL [30], or a file-based database such as PSTL, a persistent implementation of the Standard Template Library [10].

We also provide a general-purpose database interface which is a set of template classes that offer general database handling functions, along with data-types used within a database. They do not contain functions to access specific databases but are able to use concepts like SQL statements and cursors. The template classes provided are:

**DB** is used to create, maintain, and close a database connection. How this is done depends on the concrete database driver. Also, this class allows for the execution of SQL statements that do not return any data such as INSERT statements.

**DBQuery** helps formulating SQL queries. The query is written to the object as if it would be written to a stream.

**DBResponse** executes an SQL query that returns some data. Again, the implementation depends on the concrete database driver. The class has many features similar to those of a C++ STL container [24].

**DBIterator** implements a C++ STL input iterator [24] that iterates over the elements of a result set and supports database drivers with and without cursors.

**DBRow** is created when the iterator is accessed. Its index operator ("[]") is used to retrieve an actual data item. It uses either an integer (the column

id) or a string (the column designation) as the argument.

**DBObject** is the class that contains data items. It has the subclasses `DBString`, `DBInt`, `DBLong`, `DBFloat`, `DBDouble`, `DBBlob`, and `DBDateTime`. They store C++ strings, 32- and 64-bit integers, 32- and 64-bit floating point numbers, binary content, and dates.

In order to access a given database server, a concrete database driver (a traits class) for that database must be written. If the database has a C++ interface, this task is usually trivial because most work has already been done at the general-purpose database interface. A concrete database driver for the successful database MySQL is already available.

## 4 Application Development

The C++ Servlet Environment provides two approaches for writing web applications similar to those available in Java Servlet Environments: Servlets that use only C/C++ and C++ Server Pages that use C/C++ code embedded in HTML code.

### 4.1 Servlets

A Servlet is implemented as a C++ class inheriting from the `Servlet` class as shown in Figure 3. Subsequently this class is compiled into a shared object. Servlets can access the parameters and cookies that have been passed as part of the web request and output an HTML page onto their output stream.

The most important methods and attributes of the `Servlet` class are:

**service()**: This method is called for each client request that has to be processed. The default implementation calls either `doGet()` for GET or `doPost()` for POST requests. If necessary, this method may be overridden. The `service()` method's parameters are a `ServletRequest` object providing details about the current request, a `ServletResponse` object handling the servlet's response, and a `Session` object containing session information if requested.

**doGet(), doPost()**: These methods are similar to the `service()` method but handle only GET or POST requests.

**getSession()**: Requests a `Session` object from the session manager. The application server automatically executes this method if the servlet's `sessionType` is set. The method is implemented within the servlet base class.

**sessionType**: This attribute indicates whether the sandbox should take care of the servlet's session management and the type of session management

```

#include <cse/cse.h>

class HelloServlet : public Servlet {
protected:
    int counter;

public:
    HelloServlet() : counter(0) { session=false; }
    virtual ~HelloServlet() { }
    virtual void service(const ServletRequest& rq, ServletResponse& re,
                        Session* session=NULL) {
        re << "<html><head><title>Hello World</title></head>" << endl
        << "<body><h1>Hello World</h1>" << endl
        << "<p>Servlet request count: " << ++counter << "</p>" << endl
        << "</body></html>" << endl;
    }
};

Servlet* factory() {
    return new HelloServlet();
}

```

Figure 3: C++ Servlet Example

requested. Valid options are NULL for no session management, `MemorySession` for transient session management, and `DatabaseSession` for persistent session management surviving sandbox or server restarts.

**threadSafe:** Indicates whether the servlet is able to process multiple requests simultaneously. This functionality, however, is not yet implemented.

The `ServletRequest` class encapsulates information about the current request. It provides access to the HTTP request parameters using the `getParameters()` method. Such parameters may be passed as part of the URL in case of a GET request and as part of the request body in case of a POST request. The request parameters are decoded and returned as a map using the parameter names as keys.

Cookies sent with the HTTP request can be obtained with the `getCookies()` method. They are automatically transformed into `Cookie` objects. The return type is a vector of these objects. Unless a cookie has been changed there is no need to include it in the response sent back to the client.

Among other information, the `ServletRequest` class also provides the URL of the request (`getURL()`) and whether the request used the GET or POST method (`getMethod()`).

The `ServletResponse` object contains a stream to which the output of the servlet is written. For example,

```
re << rq.getURL();
```

writes the servlet's URL to the output. Other interesting attributes of this class are `contentType` which specifies the MIME type of the response and `cookies` which is an initially empty vector containing the cookies to be sent to the client.

Session data is provided through the `Session` class. Each session has a name and an ID. Together with the Sandbox name, they uniquely identify the session. The name distinguishes between different types of sessions within a single web application. The session IDs are assigned in a random order, which makes guessing them almost impossible and thus enhances data security.

The `setParameter()` and `getParameter()` methods allow a servlet to store and retrieve session data. Depending on the kind of session, these data are stored within the memory or within a database.

The session identifier must be passed on between subsequent requests to the web server. This can be done using cookies. The function `getCookie()` provides a cookie (as defined in [12]) that contains the session information. Cookies, however, do not work with all web clients. If the servlet programmer cannot rely on them, the methods `asLink()` and `asForm()` transform the session designation into a string suitable for links (in URL-encoded format) or for forms (as a hidden field).

## 4.2 C++ Server Pages

*C++ Server Pages* (CSPs) are stored in the web server's document root directory along with static HTML pages and can be identified by their `.csp`-extension. When they

```

<##vector%>
<%!vector<string> strings;%>

<% // check whether a string should be removed/added
ServletRequest::Map::const_iterator mi;
if((mi=rq.getParameters().find("remove"))!=rq.getParameters().end())
    strings.erase(strings.begin()+atoi(mi->second.c_str()));
if((mi=rq.getParameters().find("string"))!=rq.getParameters().end())
    strings.push_back(mi->second); %>

<html><head><title>TableServlet</title></head><body>
<h1>TableServlet</h1>
<p>This servlet stores strings within a table.</p>
<form method=get action=TableServlet.csp><p>Please enter a string:
    <input type=text size=64 name=string></input><input type=submit value="Go!">
</input></p></form>
<p><table border=1>
    <tr><th colspan=2>Strings entered to date: <%=strings.size()%></th></tr>
    <% for (vector<string>::iterator i=strings.begin(); i!=strings.end(); ++i) { %>
    <tr>
        <td><%=*i%></td>
        <td><a href="TableServlet.csp?remove=<%=i - strings.begin()%>">remove</a></td>
    </tr>
    <% } %>
</table></p>
</body></html>

```

Figure 4: TableServlet.csp

are requested for the first time these pages are converted into servlets and subsequently into shared objects.

CSPs are HTML documents enriched with special tags containing, among other things, C++ code. A sample CSP is shown in Figure 4. The tags used to identify C++ declarations and code are described below. Except for the first two tags, they have been designed similar to the JavaServer Pages (JSP) specification [27] to increase readability for people familiar with JSPs.

**<##sandbox%>** declares the sandbox the CSP belongs to. CSPs without this tag are executed within the default sandbox.

**<##include%>** denotes a header file to be included. As in a normal C++ program, header files provide declarations for functions and objects defined by a library. The library itself can be loaded from the configuration file. Different libraries may be used for different sandboxes.

**<%!definition%>** declares an attribute or a method.

**<%@initialization%>** indicates code to be placed into the constructor of the servlet's class. Here, attributes (those declared within this class and those inherited from the Servlet class) are initialized.

**<%code%>** executes C++ code. It is written into the servlet's `service()` function at the tag's location.

**<%=expression%>** evaluates an expression and inserts the result into the servlet's response. It may be of any type that can be written to an output stream.

**<!--comment--%>** declares a CSP comment. Unlike an HTML comment, its contents are not sent to the web browser.

The example shown in Figure 4 first includes the `vector` header file and declares a `vector` containing strings. The second block checks for the parameters passed to the CSP and based on these parameters removes or adds a new string to the `vector`. The remaining part of the servlet is used to display a form to add a new string and a table with the strings currently stored in the `vector`. Additionally, the strings are supplied with links to allow them to be removed.

After the example servlet has been deployed, our CSE translates it into a C++ servlet as shown in Figure 5. Include directives of the C++ Server Page are converted into include pre-processor macros at the beginning of the file (line 1), definitions are converted into attribute and member function definitions (line 8). Code and expression directives are used to form the `service()` mem-

```

1 #include <vector>
2 #include <cse/cse.h>
3 #include <string>
4
5 class CSPServlet : public Servlet {
6 protected:
7     virtual void print(ostream& os) const;
8     vector<string> strings;
9
10 public:
11     CSPServlet() : Servlet() { }
12     virtual ~CSPServlet();
13     virtual void service(const ServletRequest& rq, ServletResponse& re,
14                          Session* session= NULL);
15 };
16
17 // ... helper functions ...
18
19 void CSPServlet::service(const ServletRequest& rq, ServletResponse& re,
20                          Session* session=NULL) {
21     re << "
22 ";
23     // check whether a string should be removed/added
24     ServletRequest::Map::const_iterator mi;
25     if((mi=rq.getParameters().find("remove"))!=rq.getParameters().end())
26         strings.erase(strings.begin()+atoi(mi->second.c_str()));
27     if((mi=rq.getParameters().find("string"))!=rq.getParameters().end())
28         strings.push_back(mi->second);
29     re << "
30 <html><head><title>TableServlet</title></head><body>
31 <h1>TableServlet</h1>
32 <p>This servlet stores strings within a table.</p>
33 <form method=get action=TableServlet.csp><p>Please enter a string:
34 <input type=text size=64 name=string></input><input type=submit value=\"Go!\">
35 </input></p></form>
36 <p><table border=1>
37 <tr><th colspan=2>Strings entered to date: ";
38 re << strings.size();
39 re << "</th></tr>
40 ";
41 for (vector<string>::iterator i=strings.begin(); i!=strings.end(); ++i) {
42     re << "
43 <tr>
44 <td>";
45     re << *i;
46     re << "</td>
47 <td><a href=\"TableServlet.csp?remove= ";
48     re << i - strings.begin();
49     re << "\">remove</a></td>
50 </tr>
51 ";
52 }
53 re << "
54 </table></p>
55 </body></html>
56 ";
57 }

```

Figure 5: TableServlet.cc Generated from TableServlet.csp



ber function and HTML code is converted into statements sending it unmodified to the web client (lines 19–57).

## 5 Related Work

Since we have started with the implementation of our C++ Servlet Environment other developers have also recognized the need for a servlet environment for C++.

The commercial vendor Rogue Wave has developed *Bobcat* [21], a C++ servlet engine that has an API similar to that of the Java Servlet Specification [26]. Unfortunately, its evaluation license contains a non-disclosure agreement. Hence, we cannot include their product in this paper and have to assume that it is not yet ready for a production system.

*Ape Software*, an Indian company, has developed *Servlet++* [3] under a BSD-like free license, which also supports a basic form of C++ Server Pages. Unlike CSE, it does not contain a stand-alone server for the execution of the servlets but is implemented completely within an Apache module. Hence, an unstable servlet can compromise the stability of the Apache server itself. The *Servlet++* module supports C++ servlets with an interface similar to that of the Java Servlet class. Servlets are loaded as shared objects. Unlike CSE, servlets and C++ Server Pages have to be compiled manually. Also, *Servlet++* currently lacks session management, a fundamental necessity for every servlet environment, and hence we left it out of the comparison in section 6.

*C Server Pages* [9] is a servlet engine that has been designed with goals somewhat similar to those of the C++ Servlet Environment, but does not use a free license (commercial use is non-free). However, this system uses no sandboxes to encapsulate its servlets, so that it is less secure. There is currently no support for dynamic configuration, which means that each servlet has to be compiled manually (using a tool to transform C Server Pages into servlets and a C++ compiler), and the server then has to be restarted. C Server Pages is implemented as a CGI script, but the author has also built an Apache module which is, unfortunately, not yet available for download.

*Micronovae* [17] is developing a C++ Server Pages engine which works together with Microsoft's Internet Information Server (IIS). Like the previous system, it does not use the concept of sandboxes. Additionally, it only provides support for C++ Server Pages but not for servlets. Unfortunately, their download (beta version) includes no source code, so that our information about this system is solely based on our experiences with it.

The *Weblet Application Server* [29] seems to be a servlet engine for C++ developed by *Webletworks*. Unfortunately, we were unable to contact the web server of the application server's vendor for several months now

and were also unable to obtain a copy or other information about the system through other web sites.

Besides C++ servlet engines, there are numerous such engines for Java and scripting languages. One such servlet engine is the *Apache Tomcat* [2] servlet engine, a subproject of the Apache Jakarta Project. It has complete support of the JavaServer Pages and Java Servlet specifications and is included in Sun's reference implementation. Although Tomcat contains its own web server it can also cooperate with other web servers like the Apache HTTP Server. JavaServer Pages may be compiled by the built-in Java compiler or by alternative Java compilers such as Jikes.

*Jetty* [18] is a Java Servlet engine developed by the Australian company *Mort Bay*. It can cooperate with the Apache HTTP Server, but Mort Bay suggests to use the included web server. Jetty is available under a free license and offers both Java Servlets and JavaServer Pages. Like Tomcat, Jetty is configured using a set of XML files and may use alternative Java compilers for the compilation of JavaServer Pages.

*Swill* [13] is a lightweight programming library that provides a simple web server for C and C++. Unlike our servlet environment, its goal is not to provide a full fledged web server that allows the execution of multiple web applications. Instead, it allows developers to embed the web server into their own programs. This web server can be used to control the embedding application and to display the application's results using a web browser.

*Zope* [14] is a servlet environment that allows developers to use Python for servlet development. It includes a web server and a web administration front-end. Initial performance results have shown that Zope cannot compete with the top servlet engines. We assume that this is due to the performance penalty incurred by the Python interpreter. Zope is probably the right environment for Python programmers maintaining small web sites.

## 6 Evaluation

For the evaluation of the C++ Servlet Environment's performance we have implemented a benchmark suite that tests various aspects of the different servlet engines. The tests were performed with the built-in web server. All servlets were compiled using the individual engine's default servlet compiler before the execution of a benchmark.

### 6.1 Benchmarks

**Static Page Access.** In this benchmark we measure how long it takes to request a static HTML page together with 40 embedded images for 100 times. The primary goal of this benchmark is to measure the throughput of the servlet engine's web server.



**Bulletin Board System.** We have implemented a small bulletin board system that stores its entries in the file system. It allows users to create messages, read them (using a session for remembering the least recently read message), and deleting them again.

The test creates 100 messages of 320 characters each. These messages are then requested 50 times in batches of 10 messages. Finally, the messages are deleted again resulting in another 100 requests. The time taken for these 800 requests (message creation is verified with a second request) was measured. The goal of this benchmark is to check how well the servlet engine maintains the client's session state.

**Dynamic Page Access.** This benchmark uses a shopping cart servlet that we have implemented for each servlet engine.

For the measurement of this benchmark, we request the start page once to obtain the cookie containing the session information. Then, we add an item to the shopping cart, display the shopping cart, remove the item, and display the shopping cart again. The empty shopping cart page has 2048 bytes. A test run consists of 50 consecutive requests, with a total of 250 dynamic pages served. This benchmark is intended to measure the servlet engine's performance in a typical everyday situation.

**Parallel Page Access.** For this request, we used the previous benchmark and accessed the server simultaneously from a varying number of clients, each running on a different machine. We have measured the time needed by a single client to execute the same number of requests as in the previous test. The other clients in the benchmark were started before we started the client to be measured and also were terminated afterwards. The goal of this benchmark is to see how well the servlet engines can cope with increasing load.

**Mandelbrot Calculation.** This test measures the efficiency of calculation-intensive servlets by computing the Mandelbrot fractal. It accepts the size of the image, the area of the fractal to be calculated, and the maximum recursion depth as parameters. Although we support the generation of an xpm graphics file the output has been suppressed during this benchmark since we wanted to measure the "number crunching" performance only.

A test run consists of 10 accesses to the servlet, calculating a picture of the Mandelbrot set at a resolution of 1024x768 pixels and a maximum of 256 iterations per pixel.

**System Library Call.** Since a main reason for the design and implementation of the CSE was the possibility to easily integrate legacy applications into servlets, this test evaluates the inter-operation with legacy C and C++

code. For the test, we created a small servlet that calls functions from a shared library. A similar servlet might e.g. poll sensor data with high frequency in order to calculate a mean value. For C and C++ programs this is a straight-forward task. Java programs, however, have to use the Java Native Interface [16] which requires a JNI wrapper function to be written for each C or C++ function to be invoked. This benchmark measures how much performance gets lost at that interface.

## 6.2 Benchmark Results

The hardware for the performance tests consisted of two computers with AMD Duron/800 MHz processors running Linux. They were linked via a 100 Mbit/s switch in order to ensure constant network bandwidth. For the dynamic and parallel tests, we used 1-8 identical Intel Pentium II/350 MHz computers as clients, with the same server as above.

The results of our tests are shown in Table 1. We have included Tomcat as Sun's Java reference implementation, Jetty as an independent implementation in Java, and CSP and Micronovae as other C/C++ solutions. Since little information about the inner workings of Micronovae is available, we can only speculate why it performs better or worse than the other systems.

As shown by the static page access benchmark using Apache as front-end was the right choice for this benchmark. Tomcat and Jetty both did not perform as well. Although they can be set up to be used in combination with Apache, this setup is complicated. Mort Bay even recommends to use Jetty for static documents as well. In our evaluation, the Windows Internet Information Server was slightly faster than Apache on Linux. We assume that this effect is caused by the operating system.

The dynamic and parallel access benchmark, whose result is shown in Figure 6, reveals why Tomcat has become Sun's reference implementation. Both Java implementations perform much better than we would have assumed initially. Although we knew that our implementation leaves room for improvements, this was a surprise to us.

The CSE and Jetty scale equally well, but not as good as Tomcat and slightly worse than Micronovae. The CSE does not perform as well because in its current implementation the servlet server might be a bottleneck, as we have mentioned in Section 3.1. Jetty performs slower because it seems that its implementation has not been as well optimized as Tomcat's. CSP scales worst in our benchmark. Obviously, using the CGI for servlet execution is, at best, only a solution when C/C++ applications should be integrated into a web server whose performance is not an issue.

In the bulletin board system test all systems were relatively close together, which indicates that bulk transfers

Engine	CSE	Tomcat	Jetty	CSP	Micronovae
Static Page Access	35.79s	54.79s	48.20s	35.79s	35.03s
Bulletin Board System	3.96s	2.34s	3.66s	4.74s	2.40s
Dynamic Page Access	20.34s	18.83s	21.73s	24.62s	25.77s
Mandelbrot Calculation	10.69s	33.55s	33.22s	10.95s	14.86s
System Library Call	12.94s	209.39s	207.15s	14.16s	7.58s

Table 1: Evaluation Results

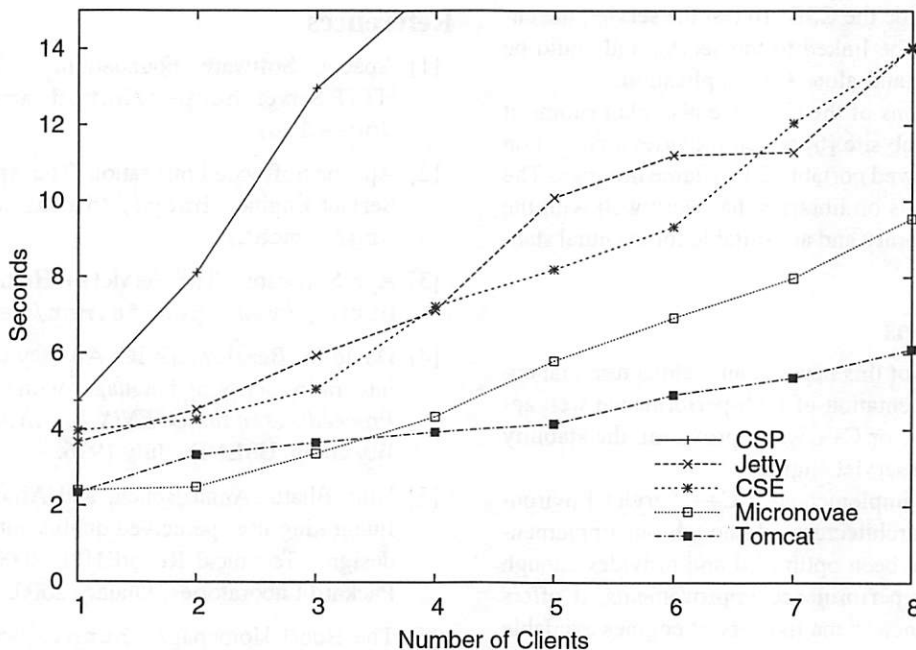


Figure 6: Parallel Page Access Benchmark Times

in Java are similarly fast as in C/C++. CSP's bad result is likely caused by the fact that it creates and initializes a new task for every servlet invocation. The difference of this benchmark over the others is that the bulletin board system's entries are stored on the file system. Hence we assume that Micronovae performs worse than the other approaches due to differences in the file system implementation between Linux and Windows.

The C/C++ based systems have a clear advantage when performing CPU intensive computations as shown by the Mandelbrot calculation. It seems that the Java just-in-time compiler is unable to optimize the code as well as the GNU compiler. In the direct comparison, the compiler from Microsoft Visual C++ which is used within Micronovae performs worse than its GNU equivalent, but we do not know what optimizations are performed.

The library benchmark shows severe limitations of the Java-based systems. In our scenario, C code is more than 16 times faster when many function calls into a legacy C application need to be done. It seems that the Java Native

Interface (JNI) which handles C/C++ library calls has not been optimized at all. CSP takes slightly more time than the CSE but is still an order of magnitude faster than the Java-based systems. The very good performance of Micronovae is probably caused by a different shared library mechanism provided by the Windows platform.

## 7 Future Work

The current implementation of the CSE has some room for optimization. This gets apparent by looking at the architecture presented in Section 3. Requests to the individual servlets are delegated to the sandboxes by the servlet server. This is a potential bottleneck and could be solved by letting the Apache module themselves delegate the requests to the individual sandboxes. Additionally, our current implementation does not yet allow the simultaneous execution of a servlet's `service()` method. This stems from the fact that we do not yet honor a servlet's `threadSafe` attribute, which results in a loss of performance.

During our tests we identified that our servlet engine is not yet capable of handling binary content correctly. We assume that this bug is located within the Apache module that passes the result of the sandboxes back to the clients. Our current assumption is that we do not handle the NULL character correctly since it indicates the end of a C string.

In future versions we also plan to implement a test environment for servlets and C++ Server Pages that supports testing outside the CSE. To test the servlet, the environment would be linked to the servlet and could be debugged like a stand-alone C++ application.

In future versions of the CSE, we also plan submit it to the BOOST web site [6] which provides a collection of free peer-reviewed portable C++ source libraries. The focus of BOOST is on libraries that work well with the C++ Standard Library and are suitable for eventual standardization.

## 8 Conclusions

The contribution of this paper is an architecture that enables the implementation of high-performance web applications using C or C++ while providing the stability known from Java servlet engines.

We have also implemented a C++ Servlet Environment using this architecture. Although our implementation has not yet been optimized and provides enough room for further performance improvements, it offers similar performance as the top servlet engines available today.

Our C++ Servlet Environment uses an API which is based on that provided by Java Servlets and JavaServer Pages. This design choice has the advantage that developers familiar with this technology will immediately be able to write C++ Servlets and C++ Server Pages.

Providing a servlet environment for C++ is important since it allows developers to reuse existing C++ code without having to use the Java Native Interface [16, 25] and hence without having to deal with different languages and the conversion of different type systems. As we have explained in Section 5, this need has recently been identified by other researchers as well. Although similar, these products are slightly incompatible to each other. Hence, we think that the standardization of a C++ Servlet Environment will be important for the future of this technology.

## Availability

The C/C++ Servlet Engine is freely available under the GNU General Public License. A more detailed description of the design and implementation of the CSE can be found in [23]. The CSE as well as its documentation is available for download from the CSE homepage at <http://www.infosys.tuwien.ac.at/CSE/>.

## Acknowledgements

We would like to thank Dave Beazley, Andreas Grünbacher, and the many anonymous reviewers for their helpful comments. We also gratefully acknowledge the financial support provided by the USENIX Advanced Computing Systems Association and by the European Union as part of the EASYCOMP project (IST-1999-14191).

## References

- [1] Apache Software Foundation. The Apache HTTP Server. <http://httpd.apache.org/docs-2.0/>.
- [2] Apache Software Foundation. The Apache Tomcat Servlet Engine. <http://jakarta.apache.org/tomcat/>.
- [3] Ape Software. The Servlet++ Homepage, 2002. <http://www.apesoft.net/servlet++/>.
- [4] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the USENIX Fourth Annual Tcl/Tk Workshop*. USENIX, July 1996.
- [5] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. Technical Report HPL-2000-3, Hewlett-Packard Laboratories, January 2000.
- [6] The Boost Homepage. <http://www.boost.org/>.
- [7] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol—HTTP/1.1*, June 1999. RFC2616.
- [8] Pankaj K. Garg, Kave Eshgi, Thomas Gschwind, Boudewijn Haverkort, and Katinka Wolter. Enabling network caching of dynamic web objects. In *Proceedings of the 12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation*. Springer-Verlag, April 2002.
- [9] Teodoro González. C Server Pages. <http://www.cserverpages.com/>.
- [10] Thomas Gschwind. PSTL—A C++ Persistent Standard Template Library. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems*, pages 147–158. USENIX, January 2001.
- [11] Arun Iyengar, Mark S. Squillante, and Li Zhang. Analysis and characterization of large-scale web server access patterns and performance. *The World Wide Web Journal*, 2:85–100, 1999.

- [12] Dave Kristol and Lou Montulli. *HTTP State Management Mechanism*, February 1997. RFC2109.
- [13] Sotiria Lampoudi and David M. Beazley. SWILL: A simple embedded web server library. In *Proceedings of the 2002 USENIX Annual Technical Conference*. USENIX, June 2002.
- [14] Amos Latteier and Michel Pelletier. *The Zope Book*. New Riders Publishing, July 2001.
- [15] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, 1997.
- [16] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, June 1999.
- [17] Micronovae. The Micronovae CSP Engine Homepage, 2002. <http://www.micronovae.com/CSP.html>.
- [18] Mort Bay Consulting. Jetty—Java HTTP server and servlet container, 2002. <http://www.mortbay.org/jetty/>.
- [19] Nathan C. Myers. Traits: A new and useful template technique. *C++ Report*, June 1995.
- [20] Netcraft. Netcraft Web Server Survey, October 2002. <http://www.netcraft.com/survey/>.
- [21] Rogue Wave Software. *The Bobcat Servlet Container: Using C++ to Integrate Applications and Business Logic with the Web*.
- [22] Peter Rossbach and Hendrik Schreiber. *Java Server and Servlets: Building Portable Web Applications*. Addison-Wesley, March 2000.
- [23] Benjamin A. Schmit. A C++ Servlet Environment. Master's thesis, Technische Universität Wien, 2002. Draft version.
- [24] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special 3rd edition, February 2000.
- [25] Sun Microsystems. *Java Native Interface Specification*, May 1997. <http://java.sun.com/j2se/1.4/docs/guide/jni/>.
- [26] Sun Microsystems. *Java Servlet Specification (Version 2.3)*, August 2001.
- [27] Sun Microsystems. *The JavaServer Pages Specification (Version 1.2)*, August 2001.
- [28] Guido van Rossum. *Python/C API Reference Manual*. Python Labs, October 2002.
- [29] Webletworks. The Weblet Application Server Homepage. <http://www.webletworks.com/>.
- [30] Michael Widenius and David Axmark. *MySQL Reference Manual*. O'Reilly & Associates, June 2002.





# U-P2P: A Peer-to-Peer Framework for Universal Resource Sharing and Discovery

Neal Arthorne, Babak Esfandiari, Alope Mukherjee  
*Department of Systems and Computer Engineering*  
*Carleton University*  
*Ottawa, Ontario, Canada*

narthorn@connectmail.carleton.ca

babak@sce.carleton.ca

alokem@cisco.com

## Abstract

We present U-P2P, an open source framework for developing, deploying and discovering file-sharing communities. We address the problem of search in peer-to-peer file sharing by allowing the end user to add metadata to shared documents. Each file-sharing community allows the sharing of a particular structured document type. Communities are themselves modeled as structured documents, thus enabling their sharing and discovery just like any other document. The creator of a particular community specifies, among other properties, the document type that it shares and the deployment model. U-P2P's extensible architecture allows developers to create new properties or extend existing ones. For example, developers can provide new deployment models or custom privacy and authentication features. U-P2P makes use of other open source projects such as Jakarta Tomcat and eXist, an XML database system.

## 1 Introduction

The current success of peer-to-peer (P2P) file sharing applications has highlighted the benefits of resource distribution and redundancy. However, to truly exploit such advantages, a few roadblocks remain to be cleared. In particular, search and discovery of resources is still quite difficult. Most known approaches rely on simple schemes such as a search for the resource name or type. File sharing communities are most efficient when the name of the file carries most if not all of the needed information. As a result, most file swapping communities are restricted to swapping music and video files. Even when the exchange of non-music files is possible, the difficulty of finding such files has been a sufficient deterrent. Lack of metadata is the main problem.

Another roadblock to more general applications of P2P has been the difficulty of creating communities for specific purposes. This arises partially from the difficulty of defining custom metadata about different types

of files. Another important problem is the current fractured state of peer-to-peer communities. Again, the lack of metadata about communities makes it difficult to know what there is to look for in the first place.

We propose a peer-to-peer framework called Universal Peer-to-Peer (U-P2P) that simplifies the sharing of custom metadata formats as well as the easy creation, configuration and discovery of peer-to-peer communities. In U-P2P, each community is described in part by the metadata of the files it exchanges. The format of a community's metadata is specified using the XML Schema language, allowing new communities centered on that file type to be created in any text editor. U-P2P allows these custom resources to be created and shared as in traditional file-sharing services.

By logical extension, the description of the community itself (the metadata format of its files, the protocol used for search, etc) is encapsulated in an XML file. In U-P2P, creating, sharing or discovering a community follows the same principles as creating, sharing or discovering a file within that community. As a result, file-swapping communities such as Napster or Kazaa can be seen as *instances* of U-P2P devoted to sharing a few specific file types and utilizing a given P2P protocol.

## 2 Related Work

Our focus is on the discovery problem in peer-to-peer systems. In Napster [1], the only files that could be shared on the network were MP3 audio files. Search was based on filenames and relied on users encoding the artist and title of each song in the MP3's title. Although metadata such as encoding rate could be used to sort the results, there was no way to search on these metadata or define other parameters. On a Gnutella [2] network, any type of file can be shared: however there is no explicit metadata handling. Search strings are passed around without processing between peers and their interpretation is left to the peer. Each peer must implement its own search algorithm using the search string

as input. Most Gnutella implementations, like Napster, simply return filenames that contain the search string. Gnutella does permit the design of overlay protocols to encode and decode metadata from search strings. Some Gnutella developers have proposed using this approach for richer metadata searches. [3, 4]. Schemas are defined for common file types: for example an audio file might be defined to have properties such as artist, title, bit rate, album, etc. The schema defines a structured format for searching MP3 metadata that is sent as a search string to other Gnutella nodes. Responding clients use the query to search local files annotated using the schema and returns the results using the same structured format. In practice though, all members of a given community must be able to speak a common language in order to communicate.

Other P2P systems such as FastTrack [5], Opencola [6] or Bitzi [7] propose variations on that idea, but they are still limited to a number of predefined schemas. The latter two have the capability to extract metadata information from a file given the file format, but this is only possible for certain formats.

Clearly, there is a need for *shared ontologies* if we want to allow search for any type of file. In the Internet world, the XML Schema format[8] is the current choice to represent ontologies, replacing Document Type Definition (DTD) [9], the schema language defined in the original XML specification. XML Schema supports the creation of custom and complex data types for XML tags, which is essential to describe aggregated or composite resources. For richer semantic descriptions, for example to allow software agents to perform search instead of humans, there can be a need to describe relationships between resources. RDF [10] and more generally the Semantic Web [11] effort address that need. The Edutella project [12] is a technology for distributed learning that uses a P2P network for sharing RDF-formatted metadata. However in Edutella metadata is the resource, not the means to describe one.

A P2P system using a semantic layering approach is not without its drawbacks. First and foremost is getting users of the system to supply metadata for their resources. The addition of metadata requires user-friendly tools for authoring RDF and schemas and a simplified approach for users who are not familiar with XML languages.

What we propose in U-P2P starts, as in Edutella, with the sharing and discovery of metadata described this using XML Schema. Once metadata is discovered it is used to instantiate a particular resource, which is in turn shared. The next section gives a high-level description of the principles behind U-P2P as well as its design.

### 3 Background: XML Schema

U-P2P relies heavily on XML Schema and the following sections include some examples of XML Schema and a brief description of the structure of XML Schema documents.

XML Schema is a Recommendation [8] by the World Wide Consortium for a schema language for XML. It constitutes a set of markup tags themselves written in XML, that are used to describe both the structure of an XML document and the datatypes of individual elements and attributes. XML Schema uses the XML Namespace Recommendation that allows authors to prefix their own tags with a specific namespace. It also allows importing types from other schemas both in the current namespace and from other namespaces.

XML Schema can be used to validate XML documents using a schema processor. XML documents themselves can link to their schema using a `schemaLocation` tag. However this is only a suggested method for specifying a schema for the document and they may use other types of schemas such as DTDs or not require validation at all.

In XML Schema, there are four basic units that can be used to describe parts of an XML document: elements, attributes, `simpleTypes` and `complexType`s.

**Elements** - Elements are the tags used in XML to label a piece of data. They have a datatype associated with them that is either defined at the same time as the element (anonymously) or is a `complexType` or `simpleType`. Elements can contain subelements, attributes and can also be part of an XML Namespace. The structure of an element can be controlled in XML Schema using sequences, choices and the same restrictions used to derive simple and complex types. For example an author can specify that an element called 'name' should consist of an exact sequence of the two elements 'firstname' and 'lastname'.

**Attributes** - Attributes are properties that belong to a parent element. They can only use `simpleTypes` and they cannot contain any elements or attributes. Attributes can be optional or mandatory within an element.

**Simple Types** - These types are either built-in data types specified in the XML Schema Recommendation or Simple Types derived from the built-in or existing types (XML Schema defines some derived types such as date and time). The Recommendation includes many useful data types such as string and decimal, that can easily be manipulated into more specialized Simple Types.

**Complex Types** - These types are a sequence or choice between a set of elements and/or attributes. They are reusable (when defined on a global level) and can be used in any part of the current schema. Once a Complex Type is defined an element can use it by specifying the

type attribute equal to the name of the Complex Type. An example of a Complex Type follows:

```
<xsd:complexType name="CanadaAddress">
  <xsd:sequence>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="postalCode">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern="w\d\w\s\d\w\d"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="province" type="xsd:string"/>
    <xsd:element name="country" type="xsd:string"
      fixed="Canada"/>
    </xsd:sequence>
  </xsd:complexType>
```

An example conforming to this schema:

```
<CanadaAddress>
  <street>1125 Colonel By Drive</street>
  <city>Ottawa</city>
  <postalCode>K1S 5B6</postalCode>
  <province>Ontario</province>
  <country>Canada</country>
</CanadaAddress>
```

Having defined the above example, the type can then be used in an element anywhere in the schema or even referenced from other schema.

XML Schema is powerful enough to describe any complex XML structures and the support for data types means the contents of shared documents are validated using the lexical representation of the data. When sharing large volumes of documents, it is useful to have full validation at the entry of the system as this will prevent the spread of unstructured data that can't be properly searched and indexed.

## 4 U-P2P Concepts

U-P2P provides four fundamental services: *search*, *create*, *browse local* and *view*. Each of these services are provided in the context of a community. We can imagine the existence of a "stamps" community that trades pictures and descriptions of stamps from around the world. On entering the stamp community the U-P2P Search function offers fields to search for stamps of a given year, and/or from a given country. Similarly, the Create function prompts you to upload a picture of the stamp, Browse Local shows the stamp objects that you have already downloaded and View displays a picture as well as the attributes for one of your downloaded stamps.

In traditional P2P applications this functionality would require downloading a client that knows about the format of a stamp object and contains customized search, create and view screens for such an object. In U-P2P, we use the power of metadata to simplify this task. Consider the following XML schema describing a stamp object:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://w3.org/2001/XMLSchema">
  <xsd:element name="stamps">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="desc" type="xsd:string"/>
        <xsd:element name="picture" type="xsd:anyURI"/>
        <xsd:element name="country" type="xsd:string"/>
        <xsd:element name="year" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

The schema above describes the expected elements of a stamp object and their data types. It is possible to generate a form from this specification using XML Stylesheet Language Transformations (XSLT) [13]. Here is an excerpt from a stylesheet that generates a *search* form from the above XML schema:

```
<xsl:template
  name="SchemaTemplate"
  match="*[local-name()='schema']">
  <h3>Search for a Resource</h3>
  <p>Enter keywords in any of the fields
  below to perform a search.</p>
  <form action="search" method="post">
    <table border="1" cellpadding="5" cellspacing="0">
      <tr><th>Property</th><th>Value</th></tr>
      <xsl:for-each select=
        "descendant::*[local-name()='element'
          and count(./child:*) = 0]">
        <xsl:call-template name="ElementTemplate"/>
      </xsl:for-each>
    </table>
    <p><input type="hidden" name="up2p:community">
      <xsl:attribute name="value">
        <xsl:value-of select="$communityName"/>
      </xsl:attribute>
    </input>
    <input type="submit" value="Search"/></p>
  </form>
</xsl:template>
```

Similarly, stylesheets can be produced that render Create forms or display a stamp object. With nothing more than a few XML documents (a schema and stylesheets for creating, searching and displaying), it is possible to define a whole new P2P file-sharing application! In fact, U-P2P provides default stylesheets for handling display and forms for resources made up of common types, making the transformation processes transparent to the novice user. Figure 1 shows the relationships between the U-P2P functions, and how they are accessed through XSL transformations.

So how can a stamp community be found or shared in the first place? The idea in U-P2P is to see a file-sharing community as just another type of resource. This is analogous to the idea of a class in object-oriented programming which specifies the structure of objects. In pure object-oriented languages such as Smalltalk, a class is merely another type of object whose structure is specified by a metaclass. Traversing the analogy in the opposite direction, a specific U-P2P community can be seen as a class instantiated by a more general metaclass: a



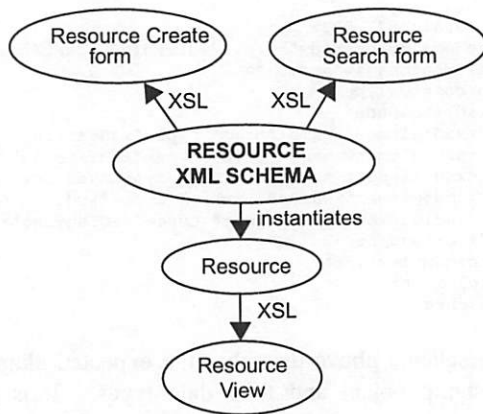


Figure 1: Generation of resource-specific displays

**Community-sharing community** (in short: a community community) **shares Community objects**:

- an *object* is an instance of its *class*, which is an instance of its *metaclass*
- *mp3* belongs to *mp3 community*, which belongs to *community community*

In U-P2P the problem of discovering the existence of a community is thus reduced to the problem of finding an object. This provides a standard way to discover the existence of resource-sharing communities.

To facilitate this, U-P2P comes packaged with one "bootstrap" schema that can be used to search for and more importantly create communities:

```

<?xml version="1.0"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="community">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="displayLocation"
          minOccurs="0" type="xsd:anyURI"/>
        <xsd:element name="searchLocation"
          minOccurs="0" type="xsd:anyURI"/>
        <xsd:element name="createLocation"
          minOccurs="0" type="xsd:anyURI"/>
        <xsd:element name="schemaLocation"
          type="xsd:anyURI"/>
        <xsd:element name="name"
          type="xsd:string"/>
        <xsd:element name="category"
          minOccurs="0" type="xsd:string"/>
        <xsd:element name="keywords"
          minOccurs="0" type="xsd:string"/>
        <xsd:element name="description"
          minOccurs="0" type="xsd:string"/>
        <xsd:element name="protocol"
          minOccurs="0" type="protocolType"/>
      </xsd:all>
      <xsd:attribute name="title" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:simpleType name="protocolType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value=""/>
      <xsd:enumeration value="Generic Central Server"/>
      <xsd:enumeration value="Gnutella"/>
      <xsd:enumeration value="JXTA"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>

```

```

</xsd:restriction>
</xsd:simpleType>

</xsd:schema>

```

As can be seen, a community can have many attributes: keywords, deployment protocol, security... This means that a community can be created by choosing specific values for such attributes (e.g. keywords: stamps, Canadian; deployment: Napster-style). The search for a community is made in similar fashion, by filling out a similar form. As of now however, we only provide one type of deployment protocol, "Napster-style", in which the peer that creates the community also acts as a broker. Other deployment models, such as "Gnutella-style" (no broker required) and "centralized repository" (a client-server option with no local copies of files) are currently being developed. This means that searching for a document could either be completely decentralized, or that on the other extreme full persistence of files could be assured by a central storage of files. Deployment decisions are thus left entirely up to the creator of the given community. Also, we have not yet explored various security or privacy schemes. Such possibilities are discussed later in this paper, in the section on design. The modular aspect of our design should hopefully allow the open source development community to provide support for many more of these attributes.

The schema combined with default stylesheets as described above allow U-P2P to become an engine for creating and searching for communities which trade all sorts of different types of files. A stamp collector using U-P2P for the first time will go to the community search form and type "stamp" into the keyword field. Upon finding a community of collectors, he or she might download the community including the community's schema and stylesheets. U-P2P then offers the option of entering the community. Once in the community the collector can perform all the actions one would expect of a file-sharing application devoted specifically to sharing stamps.

Figure 2 shows snapshots of a stamp view, a stamp search form and finally the root community view, the highest level of abstraction in U-P2P.

## 5 U-P2P Architecture and Design

Like other file-sharing services, U-P2P consists of a client and a file server running on a user's computer. The client part of U-P2P is implemented using Java Server Pages (JSPs) [15] running on a local web server. The prototype uses Jakarta Tomcat [16], but any server capable of serving JSPs may be used. The user connects to the U-P2P network by pointing their browser at the address of the local web server, typically localhost:8080.



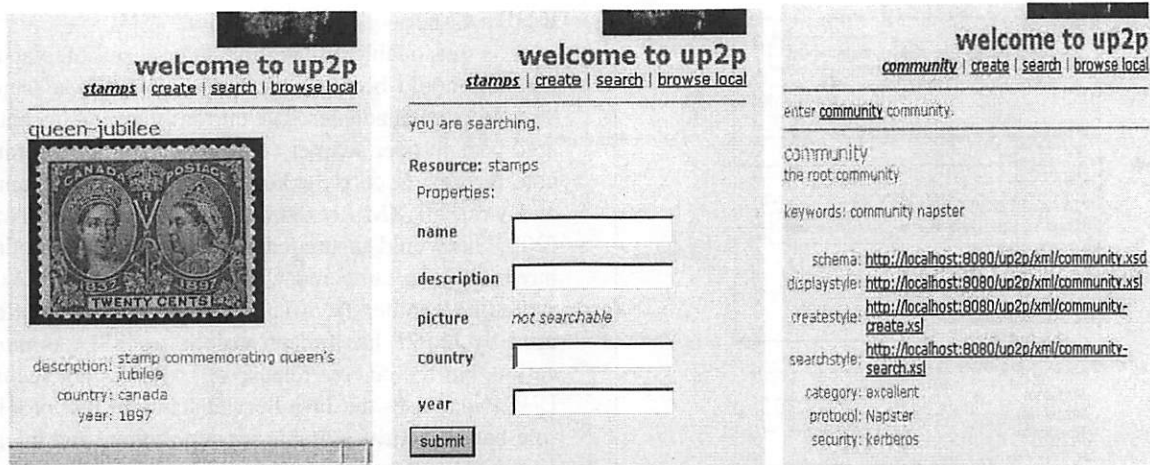


Figure 2: The "stamps" community

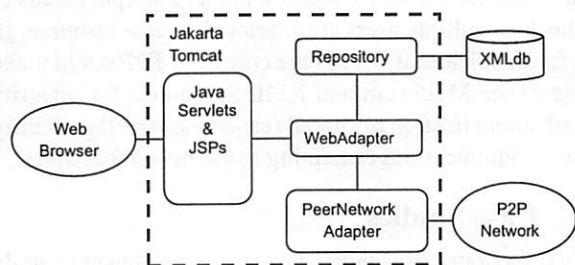


Figure 3: The U-P2P Architecture

The web server acts as GUI, and dispatches *search* and *create* requests to the other peers. In the current prototype, U-P2P follows the Napster model. This means that there is also a central server that acts as a database for information about all shared objects in the system. As with Napster, the information about the location of files is stored centrally but file transfers are conducted between peers. We are considering other possible peer configurations, such as the Gnutella distributed model and a hybrid one like FastTrack.

### 5.1 Architectural Model

U-P2P consists of three major components shown in 3: the WebAdapter, PeerNetworkAdapter, and the Repository. These components form the core of U-P2P and provide all the services needed to share and discover resources on a Peer-to-Peer network.

**WebAdapter** - Glues the components together and provides a single point of access for the user interface. This component currently supports a browser-based GUI, but could be replaced for alternate GUIs.

**Repository** - Stores all shared XML resources in a persistent XML database (using the XML:DB Database API [17]) and provides local search capabilities to the

WebAdapter and the PeerNetworkAdapter. This allows for distributed P2P topologies: each node must be able to execute searches against its own set of shared resources.

**PeerNetworkAdapter** - Provides an interface to the underlying Peer-to-Peer network and is responsible for servicing search requests, publishing resources and downloading resources from the network.

The above components are modeled as Java interfaces, with their implementations as DefaultWebAdapter, DefaultRepository and GenericPeerAdapter respectively. Additional classes include:

**FileMapper** - Maps resource IDs to real files on the local file system. When a file is 'uploaded' it is assigned an ID and mapped without modifying the file. The FileMapper is persistent in case of shutdown of the U-P2P client and file mappings are restored on startup.

**FileMapEntry** - Holds a reference to a resource file and all its attachments, pulled from the resource in the upload process. Attachment names must be unique within a resource. Resource IDs are generated from the content of the XML file using an MD5 hashing function. When hashing, a special ResourceProcessor is used that omits any attachment links within the XML. This allows the hash to stay consistent when the links are changed by another peer upon download of the resource.

**BasePeerNetworkAdapter** - A skeleton class that holds a reference to a DefaultRepository and implements the accessor for the repository. The GenericPeerAdapter is the generic P2P implementation included with U-P2P, which follows a Napster-type model. Any developer wishing to provide an alternative peer-to-peer deployment, such as a fully distributed one, or one that would plug into an existing network, would have to provide a different adapter.

**DatabaseAdapter** - Performs the dirty details needed

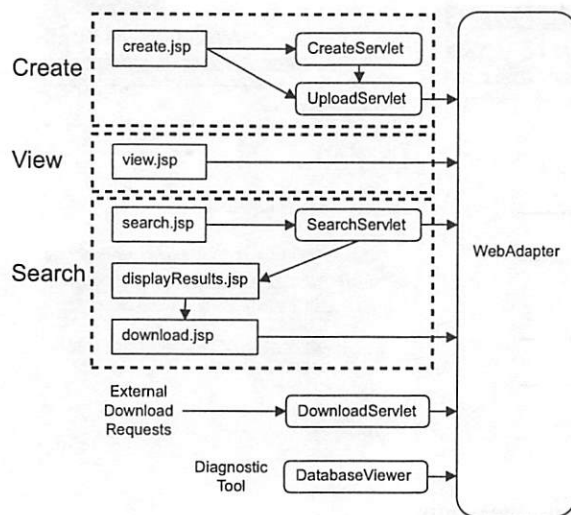


Figure 5: U-P2P Servlets

to get an XML database up and running and to configure the port that it runs on. The current implementation uses eXist 0.8, an open source XML database [18]. If a switch were made to a different XMLdb implementation, this class would be sub-classed or replaced.

The class diagram in figure 4 illustrates the relationships between these classes.

The web-based user interface requires that dynamic pages be served up to the user for such activities as *Search*, *Create* and *View*. The general flow of events that occur in one of these activities involves the user accessing a JSP, the JSP submitting a form to a Servlet and the Servlet talking to the WebAdapter and then returning through a JSP.

Figure 5 shows the pages and Servlets involved in each activity as well as the two extra Servlets needed for diagnostic reasons and for servicing download requests.

## 5.2 Security in U-P2P

In peer-to-peer systems, data integrity, authentication and authorization are concerns that are shared with other network communication systems. In U-P2P we are concerned with a layering of meta-data that is used on top of an existing network that may or may not be secure. For this reason, the bulk of security measures are left up to the network adapter used to communicate with the underlying network. Each peer network has its own requirements for security. Thus it would not be suitable for U-P2P to impose a minimum level of security for all networks using the U-P2P layering: this would restrict the ability to join public, non-secure networks such as Gnutella or Freenet. Instead, it is up to the founder of a community to decide which network adapter to use: the level of security provided by the adapter will then be

used in all network communication.

It is reasonable to assume that a set of standard adapters could be made available alongside a secured version of each adapter. The currently available centralized peer-to-peer adapter could for example, communicate through Secure Sockets Layer SSL [19] channels and wrap all XML resources with an XML Signature [20]. This would ensure that the content of the resources have not been tampered with while in transit or when shared by another user. The current Tomcat 4 platform used by U-P2P has full provisions for SSL communication, but the current release of U-P2P is not secured. U-P2P also uses the Java Servlet standard that provides role-based security suitable for deployment and integration with existing infrastructure.

It should be noted that the current implementation of U-P2P uses MD5 sums to generate a unique ID when a resource is first uploaded to the network. This ID is not intended for security purposes, but as a simple means to check if multiple users are sharing the same resource. In a future release of U-P2P, the core of U-P2P could make use of the MD5 sum and XML Signatures for integrity and authentication of shared resources, with the security of communications remaining in the network adapter.

## 6 Case Studies

Two separate case studies have been developed to study the usefulness and the performance aspects of U-P2P. The first case study attempts to solve the known problem of searching for design patterns. However, as the number of design patterns that are properly documented and formatted is not sufficient to observe the behavior of U-P2P under high load, another series of tests were run using the Genome database.

### 6.1 Pattern Repository

The Carleton Pattern Repository [21] was started in 1999. It served as a repository for software design patterns and provided extensive search capabilities over a small list of patterns. The patterns were represented in XML using a DTD designed especially for the repository project [22]. The original pattern repository project included plans to expand the repository to a distributed model.

The distributed model proposed was for each author group to have a repository server with a fixed list of the other servers in the network. The servers would form a highly distributed mesh and send out their searches to all other servers. This model was not implemented and evidently, no one else has pursued the idea.

Using the DTD as a basis, we have developed an XML Schema for representing design patterns [23]. This is used as the basis of a file-sharing community for design patterns. In addition to the schema a custom stylesheet is

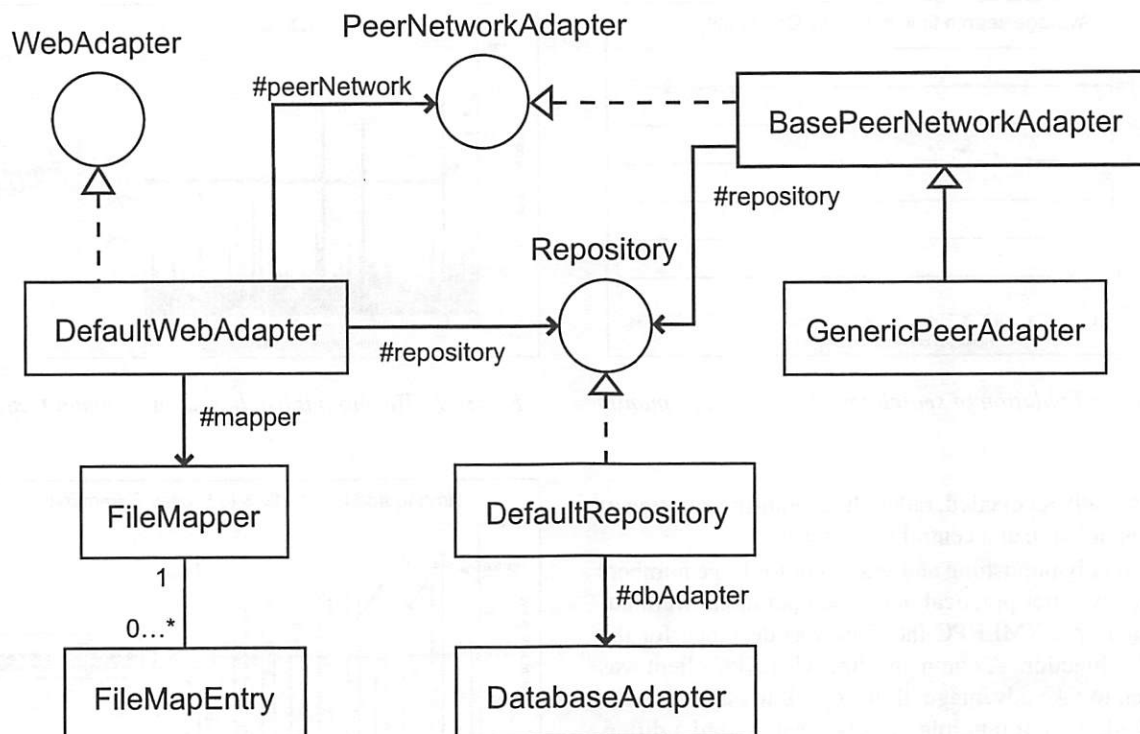


Figure 4: U-P2P core classes

required to render this complex object since the default stylesheet is tailored to simpler formats. Another design problem is deciding which parts of the design pattern should be indexed. For simplicity, our current implementation of U-P2P uploads the entire XML pattern file to make it fully searchable.

To our knowledge, prior to our work there has been no way to share design patterns in a peer-to-peer fashion that incorporates meta-data search. When fully implemented this U-P2P based system will expand the benefits of peer-to-peer file-sharing to this area. Such a system would allow computer scientists and students to publish a rich collection of patterns into an underlying peer-to-peer network, search them using rich queries and replicate popular patterns to increase their accessibility. The community-discovery aspect could also be used to access sub-communities devoted to different classes of design patterns or based on different underlying networks.

## 6.2 Drosophila Genome Project

A performance evaluation was made using a client and server running on a single computer. The performance characteristics for creating objects and searching were measured. Running on the same node removed the effect of network latency and emphasizes the delays incurred by the software and the XML database. There were two major challenges to measuring U-P2P's performance characteristics. The first challenge was hav-

ing multiple communities, each with a large collection of objects. The second was automating the process of publishing and searching for objects.

Creating a community and populating it with files manually was too time-consuming so we sought out archives containing large numbers of XML objects. One example of such a repository can be found at the Berkeley Drosophila Genome Project (BDGP) [24] This project maintains a database of annotations of different parts of the Drosophila (fruitfly) genome. The project makes these annotations available in different forms including as a large XML file. This file is split such that each annotation was stored in an individual file. Another pre-requisite for U-P2P communities is a valid XML schema. As with many existing XML repositories, the BDGP repository is described using the older DTD technology. This has been transformed into XML Schema with the help of the dtd2xs [25] package.

Although the BDGP collection was not intended for a peer-to-peer environment, it does present one possible peer-to-peer application. The BDGP is part of Fly-Base, a distributed project involving scientists at many different institutions. Each annotation contains a collection of information that can be enriched as information is gathered. With U-P2P each institution (or even each scientist) could maintain their own local database of annotations. By searching on a specific gene's name these multiple annotations could be discovered, viewed and



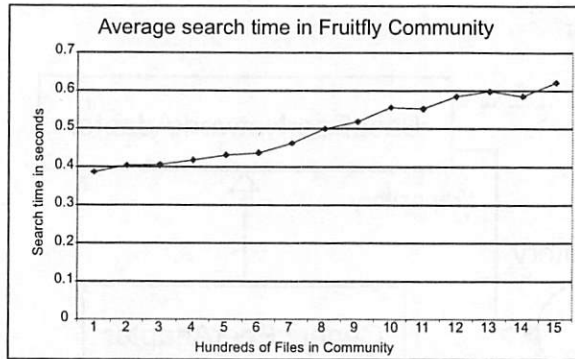


Figure 6: Evolution of search time based on community size

dynamically reconciled, rather than requiring curation of each annotation at a central repository.

Manually publishing and searching for large numbers of files was not practical, so these operations were automated. An XMLRPC interface was designed for the publish function. A command-line XMLRPC client was written to take advantage of this capability allowing the rapid addition of multiple objects. Search used a different approach, relying on the URL to encode an XPath search expression for a servlet designed to handle XPath formatted queries. These two command-line tools can then be used to script more complex experiments.

The time to publish and search was measured as repository size increases. One hundred files were published to the Fruittfly Community. Measurements are shown in Figures 6 and 7. 6 shows that publishing time increases in a linear fashion with number of files created. A set of fifty search queries was executed after one hundred files were added to the community. The same queries were repeated every one hundred files, up to fifteen hundred files. Finally each set of fifty execution times for various search queries was averaged and plotted versus the number of files. Figure 7 illustrates that the initial publishing of objects is costly: two orders of magnitude slower than the steady state time to publish. An explanation for this is the overhead required to create a new collection, a new object and whatever other internal data structures are used by the XML database. Eventually this time settles down to a steady state. On a Pentium 4 system this value was 1s.

Another interesting set of experiments was performed to determine whether publish and search times are affected by the presence of other communities. The first experiment 8 measures the time to add the first 10 files to a community as the number of existing communities is increased from none to three. Each community was created using 100 or so objects: the Fruittfly community was then created and the time to add the first 10 files recorded.

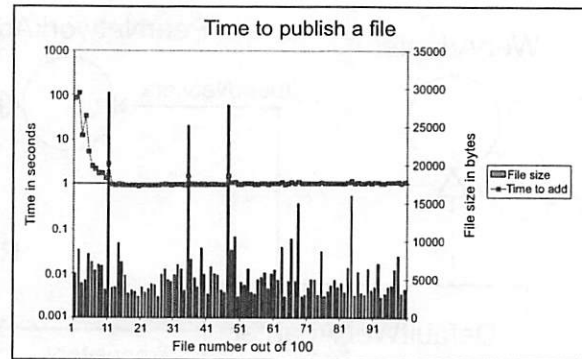


Figure 7: Time to publish based on community size

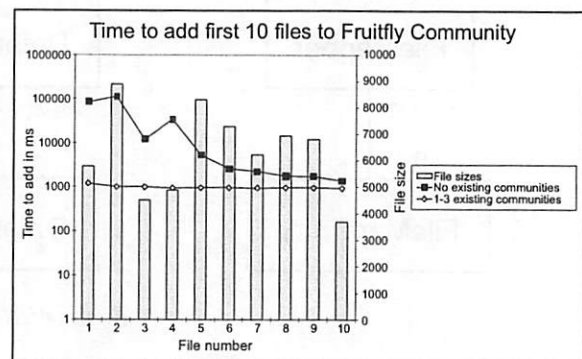


Figure 8: Time to publish when increasing the number of communities

Adding a file when there are no published objects incurs a performance penalty but these quickly converge to the 1s figure mentioned previously. Adding the first object to a community in the presence of other communities does not incur this penalty. Publishing times for objects in the new community are comparable to the time it would take to add more objects to one of the existing communities. This indicates that it is the total number of objects in the system and not the number of objects in a community which determines how long it will take to publish a new object.

In Figure 9, four searches were performed in the Fruittfly community and the time for results to return recorded. The same four searches were performed within the community in the presence of between one and three other communities (each with about 100 files). The first of the searches took about twice as long without the presence of other communities than with, but the subsequent three searches took about the same time regardless of the number of communities. This indicates that search times converge more quickly than publish times and that the existence of other communities does not affect search times.



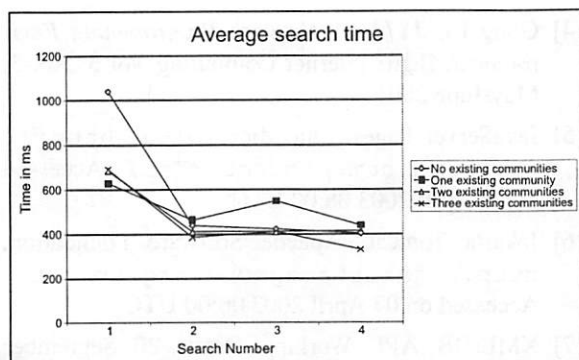


Figure 9: Search time when increasing the number of communities

### 6.3 Further Comments

A permanent U-P2P server has been set up (<http://24.102.27.174:3300/up2p>) and the Fruitfly gene annotations have been published there. In addition three other collections of XML documents have been published:

- DBLP [26] - references to Computer Science related papers, theses and proceedings
- EDGAR [27] - filings with the US Securities and Exchange Commission (SEC)
- Gene Ontology [28] - database of gene-related concepts

These are all pure XML communities: objects do not contain embedded objects. A few hundred objects have been published in each community. These can be searched by accessing the client running at the permanent location or via a U-P2P peer installed elsewhere. In addition, Chemistry Markup Language (CML) researchers also attempted to create a community for sharing molecules using this server.

In general these preliminary performance results indicate that base performance is a general problem: publish times of 1s on a Pentium 4 are too slow. However, the quick convergence of publish and search times, their slow linear growth as the number of files increase and practical experience indicate that a U-P2P node is scalable and can accommodate a few thousand files before becoming too slow to be useful. One approach to remedying this problem would be to upgrade to the latest version of XML:DB which promises performance enhancements. Another option would be to use the file system for storing documents although this makes rich search more difficult. One concrete improvement is the ability to designate other nodes to be central servers for new communities. This offloads requests as well as storage space allowing the network as a whole to serve many more objects.

## 7 Conclusion and Future Work

U-P2P is a peer-to-peer framework that allows a user to describe, share and discover communities just like any other resource. Once a community is found, its schema and associated stylesheets are downloaded and are used to perform search and publishing of resources specific to that community. Communities play a generative role similar to metaclasses in object-oriented languages. Possible applications of U-P2P include sharing resources such as resumes, knowledge management in a corporate setting, or distributed repositories for design patterns and software components.

A major direction for future work is in demonstrating the protocol independence of U-P2P. By developing PeerNetworkAdapters to interface to existing networks such as Freenet or Gnutella, U-P2P could become a meta-data layer that would provide an enhanced community-based search capability.

U-P2P communities may also be extended beyond the schema format to include definition of parameters such as protocol, security and authentication. These various parameters define a design space for peer-to-peer systems. Instead of envisioning a single system that is all things to all people, a more practical approach is to consider which methods are most appropriate to the application. The system should allow important features such as security or authentication to be decoupled and mixed and matched.

Despite some database performance issues that we believe can be overcome, as it stands U-P2P can be easily used for distributed sharing of XML documents, while exploiting structured metadata to optimize search.

### Availability

U-P2P is an open source application licensed under the GPL, and makes use of other open source products such as Jakarta Tomcat [16] and eXist [18]. Complete source code and documentation as well as guides and presentations are accessible at <http://u-p2p.sourceforge.net>.

### Acknowledgments

This work is supported by the Natural Sciences and Engineering Research Council of Canada and the Foundry Program of Carleton University. Valuable feedback on U-P2P has been received from Peter Murray-Rust and his team at the University of Cambridge, UK. The design pattern case study would not have been possible without the help of Darrell Ferguson and Dwight Deugo of the Carleton School of Computer Science.

## References

- [1] Napster, <http://www.napster.com> Accessed on 07 April 2003 06:00 UTC.
- [2] Gnutella, <http://gnutella.wego.com> Accessed on 07 April 2003 06:00 UTC.
- [3] Thadani, Sumeet, *Meta Information Searches on the Gnutella Network*, [http://www.limewire.com/index.jsp/metainfo\\_searches](http://www.limewire.com/index.jsp/metainfo_searches) Accessed on 07 April 2003 06:00 UTC.
- [4] Thadani, Sumeet, *Meta Data searches on the Gnutella Network (addendum)*, <http://www.limewire.com/developer/MetaProposal2.htm> Accessed on 07 April 2003 06:00 UTC.
- [5] FastTrack, now owned by Sharman Networks and found in Kazaa Media Desktop <http://www.kazaa.com> Accessed on 07 April 2003 06:00 UTC.
- [6] OpenCola project, <http://www.opencola.com> Accessed on 07 April 2003 06:00 UTC.
- [7] Bitzi, <http://www.bitzi.com> Accessed on 07 April 2003 06:00 UTC.
- [8] XML Schema Part 0: Primer, W3C Recommendation, 2 May 2001 <http://www.w3.org/TR/xmlschema-0/> Accessed on 07 April 2003 06:00 UTC.
- [9] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation, 6 October 2000, <http://www.w3.org/TR/REC-xml> Accessed on 07 April 2003 06:00 UTC.
- [10] Resource Description Framework, W3C <http://www.w3.org/RDF/> Accessed on 07 April 2003 06:00 UTC.
- [11] Tim Berners-Lee, James Hendler and Ora Lasila, *The Semantic Web*, Scientific American, May 2001.
- [12] Nejdli, Wolfgang et al, *EDUTELLA: A P2P Networking Infrastructure Based on RDF*, <http://edutella.jxta.org/reports/edutella-whitepaper.pdf> Accessed on 07 April 2003 06:00 UTC.
- [13] Extensible Stylesheet Language Transformations (XSLT) Version 1.0, W3C Recommendation, 16 November 1999, James Clark (Editor), <http://www.w3.org/TR/xslt> Accessed on 07 April 2003 06:00 UTC.
- [14] Gong Li, *JXTA: A Network Programming Environment*, IEEE Internet Computing Vol 5. No. 3. May/June 2001
- [15] JavaServer Pages, Sun Microsystems, <http://java.sun.com/products/jsp/> Accessed on 07 April 2003 06:00 UTC.
- [16] Jakarta Tomcat, Apache Software Foundation, <http://jakarta.apache.org/tomcat> Accessed on 07 April 2003 06:00 UTC.
- [17] XML:DB API, Working Draft, 20 September 2001, Kimbro Staken (Editor), <http://www.xmldb.org/xapi/xapi-draft.html> Accessed on 07 April 2003 06:00 UTC.
- [18] eXist 0.8, <http://exist.sourceforge.net> Accessed on 07 April 2003 06:00 UTC.
- [19] The SSL Protocol Version 3.0, Internet-Draft, 18 November 1996, <http://wp.netscape.com/eng/ssl3/draft302.txt> Accessed on 07 April 2003 06:00 UTC.
- [20] XML-Signature Syntax and Processing, W3C Recommendation, 12 February 2002, <http://www.w3.org/TR/xmlsig-core/> Accessed on 07 April 2003 06:00 UTC.
- [21] Dwight Deugo, Darrell Ferguson, Carleton Pattern Repository, <http://muffin.nexus.carleton.ca/~darrell/repo/> Accessed in July 2002.
- [22] Dwight Deugo, D. Ferguson, *XML Specification for Design Patterns*, Proceedings of the Second International Conference on Internet Computing (IC'2001): 407-412.
- [23] Neal Arthorne, *An XML Schema for Design Patterns*, <http://chat.carleton.ca/~narthorn/project/patterns/pattern.xsd> Accessed on 07 April 2003 06:00 UTC.
- [24] G.M. Rubin, *Around the Genomes: The Drosophila Genome Project*, Genome Research (1996) 6:71-79
- [25] DTD2XS, <http://puvogel.informatik.med.uni-giessen.de/lumrix/> Accessed on 07 April 2003 06:00 UTC.
- [26] DBLP, <http://dblp.uni-trier.de/xml/> Accessed on 07 April 2003 06:00 UTC.
- [27] EDGAR, <http://bulk.resource.org/edgar/xml/> Accessed on 07 April 2003 06:00 UTC.
- [28] The Gene Ontology Consortium, *Gene Ontology: tool for the unification of biology* Nature Genetics 25: 25-29.

# GNU Mailman, Internationalized

Barry A. Warsaw

*Zope Corporation*

barry@zope.com

barry@python.org

<http://www.zope.com>

<http://barry.warsaw.us>

## Abstract

GNU Mailman is a mailing list management system that has been in production use since 1998. In December 2002, a version 2.1 was released containing many new features. This paper will describe one of the most important – Mailman 2.1's internationalization support. Presented here are the tools that were built and the approaches Mailman took to marking and translating text, as well a review of some of the benefits and pitfalls of Mailman's solution. Also presented will be some future directions for internationalized Mailman, as well as other complex Python applications such as Zope.

## 1 Introduction

GNU Mailman was invented by John Viega sometime before 1997, or so is indicated by the earliest known archived message on the subject. The earliest hit for “viega mailman” in Google groups is about the Dave Matthews band mailing list that John was running [Viega97].

At the time, python.org [Python.Org] was running a hacked version of Majordomo for all its special interest group (SIG) mailing lists, but this had two problems: first, the site was becoming unmaintainable as the administrators tried to customize new features into Majordomo, and second, it just wouldn't do to run Python's mailing lists on a Perl-based list server.

When Mailman was first released, python.org quickly adopted it and has been using it ever since. Mailman 2.0 marked a milestone in its development, as version 2.0.13 is quite stable, and deployed at thousands of sites. It runs everything from small special interest group lists to huge announcement lists, at sites ranging from the commercial (RedHat, SourceForge, Apple, Dell, SAP, and Zope Corporation), to the hacker community (XEmacs, Samba, Gnome, KDE, Exim, and of course Python), to numerous educational organizations and non-profits. There are lots of hosting facilities providing Mailman services, and increasingly, quite a few international organizations.

One of the reasons for the increased interest from the non-English speaking world is that Mailman 2.1, which had been in development for about two years, is fully internationalized. Internationalization is the process of preparing an application for use in multiple locales. Localization is the process of specializing the application for a specific locale. For example, during internationalization, all end-user displayable text in the Mailman 2.1 source code was specially marked as requiring translation. Mailman 2.1.1 (the latest available patch release at the time of writing), has been localized to almost 20 natural languages.

While Mailman 2.1 may appear to be only a minor revision over 2.0.13, it really represents quite an extensive rewrite. It could easily have been argued that this version should be called Mailman 3.0. Before describing the details of the internationalization work, a brief overview of Mailman is provided, including

a quick tour of some of the other important features in the 2.1 release.

## 2 What is GNU Mailman?

“GNU Mailman” (informally referred to as just “Mailman”), is a system for managing electronic mailing lists. It is implemented primarily in Python, an object-oriented, very high-level, open source programming language. Mailing lists are administered by a list owner, and users can interact with the list – including subscribing and unsubscribing – through the web and through email. Site administrators can also interact with Mailman via a suite of command line scripts, or even via the interactive Python prompt. Mailman is the official mailing list manager of the GNU project and is available under terms of the GNU General Public License [GPL].

Mailman strives for standards compliance, and as such is interoperable with a wide range of web servers and browsers, and mail servers and clients. Of the web servers, it requires the ability to execute CGI scripts, and of mail servers it requires the ability to filter messages through programs. Apache is probably the most widely used web server for Mailman, and any of the Big 4 mail servers (Sendmail, Postfix, Qmail, and Exim) will work just fine. The HTML that Mailman outputs is extremely pedestrian so just about any web browser should work with it, as long as it supports cookies. Mailman should work with any MIME-compliant mail reader. Mailman works on any Unix-like operating system, such as GNU/Linux.

Mailman supports a wide range of features, such as:

- User selectable delivery modes. Members can elect to receive messages immediately, or in batches called digests. Two forms of digests are supported, RFC 1153 style plain text digests [RFC1153], and MIME multipart/digest style digests. Non-digest deliveries can be personalized specifically for the recipient of

the message. This means that various aspects of the message, i.e. the header or footer, or the To field, can contain information specific to the member receiving the message.

- Extensive privacy options which allow a list administrator to select policies for subscribing and unsubscribing (open, confirmation required, or approval required), policies for posting to the list (open, moderated, members only, approved posters only), and some limited spam defenses.
- Automatic bounce processing. Bouncing addresses are the bane of any mailing list, and Mailman provides two mechanisms for automatic bounce detection, regular expression based bounce matching and Variable Envelope Return Paths [VERP].

RFC 3464 [RFC3464] and the older RFC it replaces [RFC1894] describe a standard format for bounce notifications. However, many mail systems ignore or incorrectly implement this standard. For recognizing bounce messages, Mailman has an extensive set of regular expression based matches used to dig the bouncing address out of the notice. For fool-proof bounce detection, Mailman also supports VERP, a technique where the intended recipient's address *as it appears on the mailing list* is encoded into the envelope sender of the message. Because remote mail servers are required to send bounces to the envelope sender, Mailman can unambiguously decode the intended recipient's address and register an accurate bounce. Note that technically, VERP must be implemented in the mail server, but Mailman's use of the technique is close enough to warrant the label.

- Archiving. Mailman comes bundled with an archiver called Pipermail. Pipermail's chief advantages are that it comes bundled, that it is implemented in Python, and that in Mailman 2.1 it is internationalized, allowing the display of messages in alternative languages and character encodings. Its primary disadvantages are that it doesn't support searching and isn't very customizable. Mail-



man is easily integrated with external archivers.

- A mail to news gateway. Mailman can be configured to gateway lists to and from Usenet newsgroups. For example, the comp.lang.python newsgroup is gatewayed to the python-list@python.org mailing list. Even moderated lists, such as comp.lang.python.announce can be gatewayed, with Mailman serving as the moderation tool.
- Auto-responder, content filtering, and topics. The auto-responder can be set up to send a canned message whenever someone posts to the list, or emails the list owner or -request robot. Content filtering allows the list owner to explicitly filter or pass specific MIME (Multipurpose Internet Mail Extensions [RFC2045]) content types. Topics allow the list owner to assign incoming messages to any of a configurable number of groups, and members can “subscribe” to a specific topic, receiving only the subset of list traffic that matches the desired topics.
- Virtual domains. Mailman can be used on a mail server that supports multiple virtual domains. For example, the python.org and zope.org mail domains are run on the same machine, from the same Mailman installation. The one limitation in Mailman 2.1 is that a mailing list with the same name may not appear in more than one domain. This restriction will be lifted in future versions.

Mailman also provides each list with its own home page (called a “listinfo” page) which can be customized through the web. Mailing lists can be automatically created and deleted through the web (with proper support from the mail server). Mailman also provides web-based approval of moderated messages and subscriptions. There are a host of other smaller new features in Mailman 2.1 which won’t be described in this paper.

### 3 Internationalization Issues

The new features in Mailman 2.1 are extensive, but the most visible addition is the support for multiple natural languages. This means that all the administrative and publicly visible web pages, all the email notifications, and even the built-in archiver can be configured to produce text in any of nearly 20 natural languages out of the box. A large part of the re-architecting of Mailman for 2.1 has been to provide a framework for easily adding new natural languages as they become available from volunteer translation teams.

#### 3.1 Message IDs

Not every string in an application needs to be translated. For example, some strings are used as keys in dictionaries, or represent mail headers, or contain HTML tags. To make the proper distinction we refer to strings that are intended for human readability as “text” or “messages”. One of the most labor intensive parts of internationalizing an existing code base such as Mailman’s is to go through every string in the software and distinguish messages from ordinary strings. In addition to the non-translatable strings described above, the decision was made to not translate log messages since these are not intended for the end-user, and would make debugging in global community more difficult.

Each message that is to be translated needs to have four pieces of information at runtime in order to calculate the translated text: the application domain, the message id, the default text, and the target locale. Because Mailman is a fairly self-contained application, there is only one static domain, the “mailman” domain, which never changes during the life of the program’s execution.

The message id and default text are two related, but distinct concepts. The message id uniquely identifies the textual message to be displayed to the user. The message id names the message but it may not necessarily be the message. It is the message id which is the primary key into a translation catalog dictio-

nary.

The default text is the text to use as the translation of the message id, when the id is not found in the translation catalog. Because coordinating 20 different language teams is a project management challenge, it is common for some language catalogs to lag behind the source code development. Mailman releases are rarely delayed so that language teams can catch up (although advance notice of impending releases is usually given). It is often the case, therefore, that a particular message id won't be found in a specific language catalog. The default text is the fall back to use in this case.

As an example, suppose a web form had a Delete button. The message id for the button might be something like "form27-delete-button", while the default text might be "Delete".

Message ids may be explicit or implicit. In the above example "form27-delete-button" is an explicit message id. While it uniquely identifies the message to be used, it does not contain any text that will be displayed to the user. The advantage of explicit message ids is that they are immune to minor typos or formatting changes (e.g. whitespace or punctuation additions or deletions). The disadvantages of explicit message ids are two-fold: they require an extra catalog mapping message ids to the default language (e.g. English in Mailman's case), and they make the source code less readable. The latter is the more serious consequence; since nearly all human readable text in Mailman exists in Python source code, using explicit message ids would make the code nearly unreadable. A developer would have to consult the English catalog several times for some lines of code.

The alternative approach is to use implicit message ids, where the message id serves a dual purpose as the default text. Thus the human readable text that appears in the Python source code is first used as the message id, and if that fails to find a translation, it is used as the default text. While this has the advantage of making the source code more readable and easier to develop, it has several disadvantages. First, a message

such as "Delete" which has one spelling in English, may be translated to one of several different words in another language, depending on the context. This poses a problem for the translator because the message id "Delete" may appear a dozen times in the application, but may require several different words in the target language. Also, minor changes in formatting or punctuation change the message id, which requires a re-translation (this may be considered an advantage because changes in punctuation can cause semantic differences, requiring a re-translation anyway).

There is no perfect solution, but Mailman has decided to use implicit message ids because of the source code readability advantages. This occasionally requires negotiation between the application developers and the translation teams to choose appropriate and distinguishable message ids, and imposes a sort of inertia against changing existing text in the source code. One way to alleviate these problems in future releases would be to use a mix of implicit and explicit message ids, where implicit ids are used predominantly, but in rare cases explicit ids (along with a partial English catalog) are used to resolve ambiguities.

## 3.2 The Locale

Internationalizing a web-based application is much more complicated than internationalizing a command line program such as 'ls' because the natural language context (i.e. the "locale") is determined by the web request, or the email message being processed, instead of by the user's shell. In Mailman, the locale is dynamic and fluid; there may in fact be several locales needed to process any particular email message. Most of the existing techniques for internationalizing programs assume a static locale and a single domain. Mailman inherits the single domain tradition of these tools, but it uses dynamic techniques to calculate the translation locale.

We use the term "locale" and "language" interchangeably below, although this is not completely accurate. A locale describes much more than the language used; it also defines

the character encoding, as well as the formatting of dates, numbers, currency, etc. However, since Mailman does not currently support the localization of data such as dates, the language selection is the most important aspect of the active locale. Typically (although not exclusively) a single character encoding is used for a single language.

At any given point in the processing, the following locales may exist.

- The source code language. Mailman is developed in English, so by default, the English text is always available. Translations can be, and often are, incomplete and the English message is the global fall back.
- The site default language. Of the nearly 20 languages that Mailman supports out of the box, the site administrator can choose one of those languages as the “site default language”. When no other locale is known, the site default will be used.
- The list default language. Every mailing list has a default language as well as a set of alternatively supported languages. The list default language is used for all the administrative pages. It is also the language used when the list context is known and there is no overriding context.
- The page default language. The list administrator can also choose to let the list support any other language allowed by the site administrator. A user browsing the list overview page can choose to view that page in any of those languages, by selecting that language in a pop up menu on the list’s public web pages.
- The user’s preferred language. The user can also choose one of the list supported languages to be their preferred language, by making this choice in their preferences page. All email notices that Mailman sends out to the user, or any web page that the user views when logged in, is displayed in the user’s preferred language.

To support these multiple language contexts, Mailman uses an object-oriented ap-

proach where the locale is represented by an instance of a class. While this may seem natural, it is actually an elaboration of the global translation contexts in classic internationalized programs. This will be described in more detail later.

### 3.3 Character Encodings

Above and beyond the natural language issues, character encoding issues are probably the most vexing for the Mailman developers. “Character encoding” is usually referred to as the character set or charset, after the email header parameter described in RFC 2045.

A naive view would create a one-to-one correspondence between language and charset. For example, you might say that all Spanish text should be rendered in the iso-8859-1 (Latin-1) character set [ISOSoup]. However, even this simple example isn’t accurate because the Euro sign is available only in iso-8859-15.

The problem is exacerbated by some Asian languages. Japanese for example may appear in any of euc-jp, iso-2022-jp, shift-jis, and may be different depending on whether the text appears in a web browser or in an email message. In fact, Mailman 2.1’s naive approach causes some problems for Japanese users, especially when an email message is displayed as a web page in the archiver. This will be fixed in a future release.

Usually, English text uses the us-ascii character set, but for maximum interoperability, a list conducted in English may still want to be aware of Latin-1 characters. Mailman has to be careful when combining characters in different charsets, especially those for which us-ascii is not a subset.

For example, say a Spanish list received a message in Turkish, which uses Latin-5 (a.k.a. iso-8859-9). When that message is archived, different parts of the HTML page for the message will be in iso-8859-1 and other parts will be in iso-8859-9. But since HTML is inadequate at allowing multiple charsets in a single web page, the characters in one or the other



of those charsets must be converted to HTML entities, using their Unicode equivalent.

Multiple character set issues can also arise in the processing of email messages. Say for example that a message to a German list arrives in Japanese. Mailman has a feature called “headers and footers” which allow the list administrator to add some canned text to the start and end of a message (e.g. “To unsubscribe, click here”). Previous versions of Mailman would simply paste the header and/or footer around the original message body. This was broken for several reasons. The most obvious one is that if the message is really a Base64 encoded image, adding some spurious ASCII text around the original body would break the decoding. But if the message contained text in a different character set than the header or footer text, concatenation may render the original body unreadable. The solution requires careful examination of the original message, and in the extreme, ripping apart and reconstituting the structure of the original message, so that the headers and footers will always be added in a MIME-safe way.

Internationalization standards for email and HTML are defined in a series of RFCs, and these must be adhered to. For example, the most fundamental email RFC is 2822 [RFC2822] (which recently superseded RFC 822). This RFC describes the structure of an email message, but it is naive in its ASCII bias. RFCs 2045 through 2047 were added to address the use of multilingual character sets in email messages. RFC 2047 [RFC2047] was added to describe how non-ASCII characters are to be encoded in Subject fields and in other email headers. Mailman must be able to both interpret email messages with RFC 2047 encoded headers, and produce properly formatted ones when necessary. The challenge is to parse well intentioned, but erroneously encoded headers (to give the benefit of the doubt). These types of errors are all too common in email messages found in the wild and Mailman must be made robust against these types of poorly formed messages.

Prodded by these various issues, a comprehensive email package [Email] was developed

and added to Python 2.2. The email package is compliant with all the relevant MIME RFCs, as well as other mail related standards.

### 3.4 Message Catalogs

GNU gettext [Gettext] is a widespread formal model for supporting multilingual applications in traditional C applications. Gettext encourages the use of implicit message ids. This leads to a rhythm whereby the C programmer marks translatable text in the source code by wrapping them in a function call. The function is usually `_()` – called “the underscore function” – and it has both a run-time behavior and an off-line purpose. At run-time, the underscore function performs the lookup of the message id in a global language catalog. There is also an off-line tool which searches all the source code for marked strings, extracting them and placing them in a message catalog template, called a .pot file.

GNU gettext contains both a C library and a suite of tools provided by The Translation Project [TranslationProject] to manage internationalized programs. The message extraction tool is called `xgettext`. While newer versions of `xgettext` understand Python source code to some degree, a pure-Python version of the program called `pygettext` was developed and is distributed with Python. `pygettext` has some additional benefit, including the ability to extract Python docstrings which may not be marked with the underscore function.

Mailman has adopted the gettext model of marking and translating source strings, and to that end, a GNU gettext-like standard module was implemented for Python [GettextModule]. While the gettext module implements the same global translation model of the C library, two elaborations were necessary for a more Pythonic interface.

First, for long running daemon processes such as Mailman 2.1’s mail processor, multiple language contexts are required, so the global state implied by gettext isn’t always appropriate. Here’s an example to illustrate understand why.



When a new member subscribes to a mailing list, two notification messages can be sent. One is a welcome message sent to the member, and the other a new member notification sent to the list administrator. If the list's preferred language is Spanish, but the user prefers German, these two notifications will be sent out in two different languages. Since a single process crafts and sends both notifications, simply using `_()` wrapping doesn't give enough information. Which language should the underscore function translate its message id to?

Python solves this problem by providing an object-oriented API in addition to gettext's traditional functional API. Using the object interface, a program can create instances which represent the translation context; in other words, a single target language catalog is fully encapsulated in an object. For convenience, this object can be stored in some global context, and in the Mailman source, this global object can be saved and restored as necessary. Here is a simplified Python example:

```
# The list's preferred language is in
# effect right now
saved = i18n.get_translation()
try:
    i18n.set_language(
        users_preferred_language)
    send_user_notification()
finally:
    i18n.set_translation(saved)
send_admin_notification()
```

The second problem might be termed syntactic sugar or simple convenience, but it turns out to be extremely important in a Python program filled with translatable text. Python strings support variable substitution (also called "interpolation"), whereby a dictionary can be used to supply the substitutions. For example:

```
listname = get_listname()
member = get_username()
d = {'listname': listname,
     'member': member,
     }
print _('%(member)s has been '
        'subscribed to %(listname)s') % d
```

This is a critically important feature for internationalized programs because some languages may require a different order of the substitutions to be grammatically correct. While stock Python supports this requirement, its implementation leads to overly verbose code. In the above example, we've written the words "listname" and "member" four times each. Now imagine that level of verbosity duplicated a hundred times per source file. "Tedious" comes to mind!

Mailman solves this by providing its own underscore function, which wraps the gettext standard function, but provides a little bit of useful magic by looking up substitution variables in the local and global namespace of the caller. Using Mailman's special underscore function, the above code can then be rewritten as:

```
listname = get_listname()
member = get_username()
print _('%(member)s has been '
        'subscribed to %(listname)s')
```

While the average Perl programmer might ask what all the fuss is about, the Python programmer will notice something interesting: there's no interpolation dictionary and no modulus operator. The dictionary is created from the namespaces of the caller of the underscore function, which contains the "listname" and "member" local variables. The trick is that the underscore function uses a little known Python function called `sys._getframe()` to capture the global and local namespaces of the caller of underscore. It then puts these in an interpolation dictionary, with local variables overriding global variables, and then applies the modulo operator to the translated string, using this dictionary.

Marked translatable texts are used all over Mailman, and we run `pygettext` over all the source code to produce a gettext compatible mailman.pot catalog file. To translate this to a new language, the translation team would start by copying mailman.pot to `messages/xx/LC_MESSAGES/mailman.po` where "xx" is the language code for the new language. From here, standard tools such

as po-mode for Emacs or KDE's kbabel can be used to provide translations for all the source message ids. Then, standard gettext tools can be used to generate a mailman.mo binary file, which Python's gettext module can read. In this way, internationalized Python programs can leverage most of the tools translation teams normally use for C programs. Translators don't have to learn new tools just to translation Python programs.

### 3.5 Templates

While gettext style message text in Python source code are essential for an internationalized Mailman, they aren't always appropriate. For example, Mailman has always used templates as a way of conveniently representing full web pages or parts of email messages. These templates provide an easy way for site administrators to customize the look of the Mailman web pages, or the text sent out under various circumstances.

In an internationalized Mailman, the templates serve another purpose: they serve as a mechanism for providing language specific versions of the templates. Mailman uses almost 50 templates for various purposes, and of course provides the English versions of the templates as a default. Each supported language provides its own version of the templates, and Mailman has a defined search order for template lookup. For example, if Mailman were to display the public list overview page for a mailing list, it would search for the listinfo.html page, in the following locations (relative to the installation directory):

- The list-specific language directory `lists/listname/language/template`
- The virtual domain-specific language directory `templates/list.host.name/language/template`
- The site-wide language directory `templates/site/language/template`
- The global default language directory `templates/language/template`

The first location to yield the desired template wins. Thus, as with the gettext catalogs, English is always an available fall back.

Templates, like marked translatable source code text, support variable substitutions, using the same syntax. With templates, an explicit substitution dictionary is always provided, and the interpolation is performed after the template is located.

While the template system works well enough, its coarseness is a serious drawback. For example, say a new feature required the addition of an HTML button on one of the templates. While this is trivial to do for the English template, changing the English template means all the other templates are out-of-date. The translation teams must follow up with new versions of the modified templates, or other languages will lag behind the English version.

One solution for templates might be something like Zope Page Templates (ZPT) [Pelletier], and specifically, internationalized ZPT [I18NZPT]. Internationalized ZPT combines the best of gettext and templates by allowing the template author to design the template, marking sections of the template as translatable text. Another extraction tool can then run over the ZPT file and add the translatable messages to the overall catalog. This has the huge advantage that structural changes to a template don't require the translation teams to do any work. Changes to content messages in the template simply mean that one phrase may be out of date, but the whole template won't be invalidated.

## 4 Unicode

Python has two types of string objects, traditional 8-bit byte data strings and Unicode character strings. Python also has literal forms for each string type; quoted text are defined to be 8-bit strings unless the leading quote is prefixed with a "u", in which case it is a Unicode string. Because strings can come into Mailman in a variety of ways (e.g. through the web, an email message, or a

message catalog), the code must be prepared to handle encoded 8-bit strings and Unicode strings. Encoded 8-bit strings must be converted to Unicode via the `unicode()` built-in function in order to properly combine strings using concatenation or interpolation. In addition, Unicode strings must be re-encoded when printing them to certain streams, such as the log files, or standard output, but these encoding operations must watch out for unsupported characters. For example, if a Unicode string containing Latin-1 characters is printed to an ASCII-only terminal, an exception can be raised due to the non-ASCII characters in the string.

There is no doubt that character conversion issues have been the thorniest and most common bugs reported on Mailman 2.1 to date. While many issues have been fixed, the most important lesson learned is that Mailman should convert all text (not necessarily all strings!) to Unicode at the earliest possible time, ideally when the text enters the system. Mailman should use Unicode strings everywhere internally, converting to encoded 8-bit strings only where needed, and only at the last possible moment. Analysis will still be needed to decide how to handle conversion errors, such as those described above. In Python, the conversion function can be given an additional argument which specifies how strict the conversion should be, e.g. raise an exception if there are illegal characters found, throw the illegal characters away, or substitute a question mark for any illegal characters. The exact choice of the strictness flag will be dependent on the context in which the conversion is occurring.

## 5 Other Issues

There are some operational issues that need to be addressed for an internationalized application such as Mailman. Care must be taken when marking the source code for translation so that the text is split in a grammatically clear way. For example, whenever possible full sentences should be used, since translating sentence fragments may not be possible in all languages. Also, plural

forms and genders pose particularly thorny problems. Python 2.3's `gettext` module supports plural forms, but only alpha releases of Python 2.3 have been made available as of this writing. English doesn't have gendered nouns, and sometimes, English text source strings need to be rewritten to accommodate translators.

Python supports a number of specific character encoding "codecs" in the standard distribution. While Python has built-in support for most Western codecs, Asian codecs in particular are not supported. Fortunately Japanese, Korean, and Chinese codecs are available as third party distributions.

Internationalization is a lot more than simply translating strings; many other values from currencies to dates must also be localized if they are to be displayed correctly for a particular language or country. Long term goals include wrapping IBM's ICU library [ICU] in Python.

While internationalization imposes some performance overhead, the effect is negligible. In an application such as Mailman, the performance of the mail server that Mailman feeds messages to, the network bandwidth, and the performance of the operating system and file system have a far greater influence on the performance of the system than does the Mailman software. Internationalization has imposed no perceived performance penalty.

Internationalization has increased the size of the software distribution, since by default the download contains the message catalogs for all supported languages. The current catalog contains over 1200 message ids and is approximately 228 KB in size. The translated and compiled catalog files are from 80 to 300 KB in size depending on the completeness of the translation. In all, the message catalogs themselves add approximately 16 MB to the uncompressed program source code. The templates add about another approximately 3 MB. For this reason, future releases of Mailman may provide an English-only distribution, with separately downloadable language packs.

## 6 Examples

Here is some sample Python code (reformatted for this paper) taken from Mailman 2.1 which shows marked messages:

```
label = _(categories[category])
realname = mlist.real_name
doc.SetTitle(
    _('%(realname)s Administration '
      '%(label)s'))
doc.AddItem(Center(Header(2, _('
  %(realname)s mailing list '
  'administration<br>%(label)s '
  'Section')))))
```

Notice first that the local variables “label” and “realname” are referenced in the default text, and that their values come from the magic interpolation described above. In a translated message the order of the substitutions may change, so this ensures that the substitutions will occur in the grammatically correct location in the translated message. Also notice that there are actually three uses of the underscore function. In the second and third, the only function arguments are strings (in Python, adjacent strings are concatenated by the lexer). The `pygettext` rule for message extraction is a single string inside the underscore function, so both these texts will be extracted into the message catalog.

The first use of the underscore function is interesting in that it is getting a value out of a dictionary lookup. This is an example of a deferred translation, where the underscore function is only used for its run-time behavior. The text returned by the dictionary lookup is translated in a normal fashion, but the program source code `categories[category]` isn't extracted into the catalog because it isn't a string.

At the place where the categories dictionary is defined, the strings are also wrapped in an underscore function for `pygettext` extraction, but they aren't translated at that place in the program. We do this in a number of situations, such as when dictionaries are defined in module global scope. In that case, you would see something like:

```
def _(s): return s

categories = {
    'cat1': _('Privacy'),
    'cat2': _('Autoreplies'),
    'cat3': _('Topics'),
}

_ = i18n._
```

Here, we're marking three strings for extraction, but we aren't translating them at the point of definition, because the target locale isn't known at this time.

Due to space limitations, sample web pages can't be included, however a public internationalized list can be viewed at <http://mail.python.org/mailman/listinfo/playground>. You can view this listinfo page in any of the languages supported by Mailman.

## 7 Future and Related Work

Internationalized Mailman servers are in deployed use around the world, and many of the earliest related bugs have been satisfactorily fixed. However the basic architecture used by Mailman may undergo additional refinement in future releases. In particular, Mailman will be rewritten to use Unicode internally for all human readable text. The templates used in Mailman will likely be redesigned to use something like Zope's ZPT, which allow finer grain control over the evolution of the templates.

The experiences learned during the Mailman internationalization effort have been carried forward to the Zope 3 internationalization effort [Zope3]. Zope is a web application server and framework, also written in Python. Zope's internationalization efforts are made more complicated by the fact that it is a framework supporting multiple applications rather than a single application. This means that while Mailman needs only a single application domain, Zope may have multiple simultaneous domains, even in a single translation context. Zope therefore needs a way to record the domain that a particular message



has come from so that it can be looked up in the proper catalog at output time. The solution has been to create a MessageID object, which is a subclass of string that contains the domain as an instance variable.

Also, in Zope few translatable messages are found in Python source code. The predominant carrier of human readable text is the ZPT. Thus the mechanisms described above for simple interpolation and global translation contexts aren't appropriate for Zope. While Zope's internationalization efforts are built on the Python tools developed during Mailman's internationalization, they will ultimately improve or expand on these tools.

As part of the Zope 3 internationalization effort, a Translation Web Service (TWS) has been proposed [ZopeTWS]. While largely only science fiction at the time of this writing, the TWS is a vision of how to coordinate the project management issues related to internationalization. One of the most difficult ongoing problems for an internationalized project is coordinating the output of the software developers with the translation efforts of the language teams. While the Translation Project attempts to automate and coordinate much of this process, the TWS plans to take this aspect a step further by providing a global web service for truly collaborative translations. With the TWS, a project manager could upload message templates for particular domains, and language teams would translate messages at their own pace. When a new version of the software is prepared for release, the project manager could then download a snapshot of the current state of the various translations for the project. The key advance with the TWS is that once the infrastructure is in place, the software developers are no longer bottlenecks in the translation effort, and coordination among translation team members is automatic.

## 8 Acknowledgments

The author would like to thank Zope Corporation (<http://www.zope.com>) for their support of this work.

Mailman was originally invented by John Viega, and at various times has been shepherded, maintained, and developed by Thomas Wouters, Ken Manheimer, Harald Meland, and Scott Cotton. The author is the current project leader.

Juan Carlos Rey Anaya and Victoriano Giralt produced the first working prototypes of an internationalized Mailman, and worked with the author to design the architecture for supporting internationalization. Others who provided invaluable contributions for the internationalization effort include Ben Gertzfield, Martin von Loewis, Simone Pinno, Daniel Buchmann, Tokio Kikuchi, and Ousmane Wilane. The ACKNOWLEDGMENTS file that comes with the Mailman source distribution contains a detailed list of contributors.

## 9 Availability

GNU Mailman is free software, covered by the GNU General Public License. It is available for download from <http://sf.net/projects/mailman>

More information on Mailman can be found at its home page <http://www.list.org>

Mirrors of the Mailman site are at <http://www.gnu.org/software/mailman> and <http://mailman.sf.net>

## References

[Viega97] <http://groups.google.com/groups?q=viega+mailman&hl=en&lr=&ie=UTF-8&scoring=d&start=60&sa=N&filter=0>

[Python.Org] *Python Home Page*, <http://www.python.org>

[GPL] Free Software Foundation, *GNU General Public License*, <http://www.gnu.org/licenses/gpl.txt>

- [RFC1153] F. Wancho, *Digest Message Format*,  
<http://www.faqs.org/rfcs/rfc1153.html>
- [VERP] D. J. Bernstein, *Variable Envelope Return Paths*,  
<http://cr.yp.to/proto/verp.txt>
- [RFC3464] K. Moore and G. Vaudreuil, *An Extensible Message Format for Delivery Status Notifications*,  
<http://www.faqs.org/rfcs/rfc3464.html>
- [RFC1894] K. Moore and G. Vaudreuil, *An Extensible Message Format for Delivery Status Notifications*,  
<http://www.faqs.org/rfcs/rfc1894.html>
- [RFC2045] N. Freed and N. Borenstein, *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*,  
<http://www.faqs.org/rfcs/rfc2045.html>
- [ISOSoup] *The ISO 8859 Alphabet Soup*,  
<http://czyborra.com/charsets/iso8859.html>
- [Gettext] GNU *gettext*,  
<http://www.gnu.org/software/gettext/gettext.html>
- [TranslationProject] *The Translation Project Site*,  
<http://www.iro.umontreal.ca/contrib/po/HTML/index.html>
- [GettextModule] *Multilingual internationalization services*,  
<http://www.python.org/doc/current/lib/module-gettext.html>
- [RFC2822] P. Resnick, *Internet Message Format*,  
<http://www.faqs.org/rfcs/rfc2822.html>
- [RFC2047] K. Moore, *MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text*,  
<http://www.faqs.org/rfcs/rfc2047.html>
- [Email] *email - an email and MIME handling package*,  
<http://www.python.org/doc/current/lib/module-email.html>
- [Pelletier] M. Pelletier and A. Latteier, *The Zope book*, Chapter 5, Using Zope Page Templates, ISBN 0735711372.
- [I18NZPT] <http://dev.zope.org/Wikis/DevSite/Projects/ComponentArchitecture/ZPTInternationalizationSupport>
- [ICU] *International Components for Unicode*, <http://www-124.ibm.com/icu/>
- [Zope3] *Welcome to the Zope 3 project*,  
<http://dev.zope.org/Wikis/DevSite/Projects/ComponentArchitecture/>
- [ZopeTWS] *Translation Web Service*,  
<http://dev.zope.org/Wikis/DevSite/Projects/ComponentArchitecture/TranslationWebService>

# ASK: Active Spam Killer

Marco Paganini

paganini@paganini.net

www.paganini.net

## Abstract

We present Active Spam Killer (ASK), a program that attempts to validate unknown senders before allowing delivery of their message. Validation occurs by means of a challenge reply sent to senders who are not yet known to the system. Messages are kept in a queue pending confirmation until the sender replies to the challenge. Further messages coming from confirmed senders are delivered immediately. In a sample of 1000 spam mails, ASK was 99.7% effective at blocking spam, resulting in only 3 spam messages being delivered. Other programs' best ratios were 97.8% or as many as 22 spam messages delivered.

## 1 Introduction

*Unsolicited Commercial Emails* (UCEs), also commonly called "spam" represent a serious problem to most Internet users, who are constantly bombarded with all sorts of scams, promotions, and offensive material. This situation has prompted the creation of many different tools to eliminate the problem, with varying degrees of success.

The most common way to deal with spam is to parse incoming mails and decide whether they should be delivered or not based on their contents. Reasonable results have been obtained with this technique, but the complexity and diversity of human languages make it a difficult task. Such *content filtering* tools propose to attack one of the weak spots in spam: the message content itself.

The effectiveness of such tools can be drastically reduced when the incoming mail employs an unknown language or even an unknown character set. Also, cleverly crafted emails may never be detectable as the difference between those and perfectly valid emails is subtle.

*Active Spam Killer* (ASK) proposes to attack a different weak spot in the spam chain: the validity of the sender's email address. When a message from an unknown origin is received, a challenge (also known as *confirmation message*) is sent back to the mail originator. This message contains brief instructions to the sender on how to get authenticated into the system and cause delivery of the original message. The confirma-

tion message is crafted in such a way that a simple reply keeping the "Subject" line intact will suffice. The confirmation message also contains a unique MD5 [16] hash computed by combining the contents of the original email with a secret key known only to the recipient. This prevents false confirmation returns as the code is based on the unique characteristics of the receiver.

The message remains stored in the *pending mail queue* until a *confirmation return* is received (a reply to the confirmation message with the MD5 hash in the "Subject" header). When that happens, ASK checks the pending mail queue for a message whose MD5 signature matches the one in the confirmation return. If found, the original message is delivered from the pending mail queue and the sender's email address is automatically added to the whitelist. This sender has now been validated and all future messages from this address will be immediately delivered.

The sender's address can also be added to two other lists: the *ignorelist* and the *blacklist*. The first causes emails to be silently ignored while the second not only ignores the email but also sends a message back to the originator explaining that future emails coming from this address are blocked.

This *challenge-authentication* scheme guarantees that delivered emails always come from valid senders. Unwanted (but technically valid) senders can be easily ignored as they offer a simple key to their detection: their own email addresses.

Although an auto-responder could be used to defeat this method, this is unlikely as it would expose the sender to legal complications, account and service cancellations, and a fairly large number of "Invalid Address" response messages during normal operation.

The rest of this paper is organized as follows: Section 2 provides background information on popular anti-spam techniques. Section 3 details ASK's design and operation. Section 4 compares the effectiveness of ASK against other anti-spam tools. Section 5 surveys other challenge-based solutions. Finally, Section 6 presents our conclusions and future work.

## 2 Background

Compared to the traditional method of mailing printed propaganda, spam costs almost nothing. No mailing infrastructure is needed except for an Internet account and a personal computer. A one or two percent return on investment potential applied to millions of emails results in a number large enough to justify the approach to a number of people.

The first ingredient is a large number of email addresses, usually obtained by harvesting addresses off the Internet. A study conducted by the Federal Trade Commission concluded that 86% of all addresses posted to Web pages and to the newsgroups received spam [5]. Anything resembling an email address is collected and used. Those without the resources to mine the addresses themselves can purchase CDs containing ready-to-use lists of addresses.

Next, a *Mass Mailer* like DynamicMailer [3] is used to deliver the message to the list of addresses. These programs normally provide certain facilities like header forging, sending emails directly from the user's workstation (bypassing the need for an SMTP server) and others.

Spammers must always act as inconspicuously as possible to avoid account cancellation, revenge from angry users, and legal problems. The sender's email address is often forged to an invalid address or to a throw-away account from a free web-mail provider. These accounts are rapidly filled with bounce messages as a result of outdated and invalid emails in the list.

Instructions on how to obtain the product or service offered are normally contained in the body of the mail, with URLs pointing to the Web page of the seller.

Next, we outline the most common anti-spam techniques in use today.

### 2.1 Realtime Blackhole Lists

Misconfigured mail servers will relay any incoming message to any destination without control. These servers can be used to deliver mail indiscriminately and in many cases, anonymously.

*Realtime Blackhole Lists* (RBLs) are online databases containing the IP address of such servers. These databases are queried by *Mail Transfer Agents* (MTAs) upon receipt of a new message. If the IP address originating the connection is listed in the database, the connection will be terminated and the appropriate error code will be sent. Some RBLs also keep dial-in IP addresses in an attempt to detect messages sent directly from dial-up workstations.

Many such databases exist today, with *MAPS* [9] being a well known provider of such services. Most MTAs support this feature with minimal configuration effort.

RBLs are not very effective if used as the only spam prevention method as they can only block mail (spam or

not) coming from spammers who use open relays to send their messages. Even though some RBLs specifically list dial-in lines, often it is not practical to use those as many legitimate users send emails directly from their workstations over dialup lines.

Some RBL providers have complicated procedures to remove entries from their databases, leading to a substantial number of false positives (See Section 4.1). Also, there are normally no whitelist mechanisms available to regular users, meaning that it is impossible to grant access to a certain known valid email or IP without supervisory privileges.

### 2.2 Content Filtering Tools

Tools in this category try to detect spam by investigating the content of received emails.

Early attempts used simplistic keyword filters, normally implemented with generic mail processing tools. Specific strings like "Dear Sir" in the body of the email would flag the message as spam. This produced inaccurate results and a fairly large number of false positives.

A more sophisticated approach to the problem employs *scoring* of certain keywords, sentences and characteristics pertinent to spam mail. A simple "Dear Sir" might indicate that we are dealing with spam, but "Dear Sir," "Click Here," and "Call now" in the same email message are a very clear indication. SpamAssassin [20] utilizes this technique.

Usually, a complex set of rules exists with a positive or negative numeric score assigned to each rule. Rules that clearly indicate spam receive high values. Negative values reduce the probability of the email being classified as spam (even in the presence of other clear indications). A rule to match a string like "Dear Sir/Madam" will have a lower score than "Work from home" as the latter is a clearer indication of the common home employment Internet scams. Expressions like "Usenix" and "Algorithm" would receive a negative value as they strongly suggest that this is not a spam mail. The mail will be marked as spam if the sum of all matching scores exceeds a user-defined threshold.

There are specific and distinct rules for the email headers and body. The rules applied to the headers will be equally effective, regardless of the language used to send the email. Body analysis however is greatly affected by language: a set of rules that perfectly detects emails in the English language could completely miss emails written in Italian or Korean.

A new variation of this technique, employing a naïve Bayes classifier to isolate junk mail has become quite common [17]. Bogofilter [15] is a popular tool in this category.

These tools can be trained to learn about good and bad combinations of tokens and their probabilities of oc-



curing together in incoming mails. As they learn more about what is spam and what is not, the chances of correctly detecting junk emails increase.

Bayesian classifiers suffer from the same problem as the other content filtering approaches: a well-crafted message body is likely to pass unharmed, as few identifiable elements are present. Messages without a text body (for example, advertising as an attached image file) might pose a problem too since very little textual content is available for analysis [10].

To illustrate how difficult it is to classify e-mails as spam based solely on content, consider the following email:

```
From: abcd12345@domain.com
To: yourname@yourdomain.com
Subject: Check this out.
```

Hi,

```
I found a tool called ``CleanUp`` at
(http://www.example.com). It clears
the contents of the browser cache
and history lists so other users
won't know the websites you have
been browsing. Very useful!
```

This could represent either a perfectly valid email (if coming from a known sender) or a clear indication of spam (if coming from an unknown sender). Analysis based solely on content is very difficult in this case, especially for someone without the knowledge about any previous association between the recipient and the sender.

## 2.3 Distributed Anti-Spam Networks

Spam messages are typically sent unaltered to a large number of recipients. Distributed Anti-Spam Networks attempt to curb spam by preventing its propagation. Vipul's Razor [14] and the Distributed Checksum Clearinghouse (DCC) [18] are examples of programs using this technique.

Agents installed on the user's workstation will generate *fuzzy signatures* of every incoming mail. These signatures ignore small changes in the text so that slight variations in the spam body or headers will still generate similar signatures.

Next, a centralized database is queried looking for the signature computed from the incoming mail. If a match is found, the email is assumed to be spam and is discarded.

Users are responsible for reporting spam to the database (normally by means of a special account where the spam mail should be forwarded). Once spam has been reported by one user, all others will be protected against that specific spam mail or similar ones.

Under normal circumstances, the chances of false positives is very small, as actual users report spam to the system. Also, the chances of catching spam that is already in the database is quite good.

This approach, however, can only identify spam once a previous (or similar) case exists, meaning that newly sent spam will not be detected. A reasonable number of spam reporters is required to make the system work reliably and the quality of the database is largely dictated by the quality of the reports. Invalid or incorrect reports could poison the database, causing valid emails to be reported as spam. There is also a scalability concern, given the centralized nature of the database servers containing the signatures.

Another point to note is that an extra TCP connection is needed to contact the database servers, making this a non-viable alternative for users behind restrictive firewalls.

## 2.4 Challenge-Based Authentication

Challenge-authentication agents work on the premise that mail should only be delivered after senders identify themselves. ASK, TMDA [11], and QConfirm [13] are examples of programs that employ such a technique (with variations). When a new mail is received, the sender is checked against a database of known addresses. If the sender is known, the email is delivered immediately. Otherwise, a challenge mail will be sent back to the originator requesting confirmation. Once the sender becomes known to the system, further messages coming from the same address will be immediately accepted.

This method exploits the fact that most spammers use invalid return addresses in their messages. Since no (or limited) text analysis is performed, it is impossible to force delivery of the message by crafting the message body to look like an innocent message.

Unlike the previous alternatives, in challenge-response systems, subsequent action is required from the sender to deliver the email. Spammers utilizing valid email accounts could reply to the challenge and get authenticated to the system. Special provisions exist to avoid mail-loops or sending challenges to mailing-lists.

## 3 Design

We designed ASK to be simple to install and readily available to the widest possible audience. Supervisory rights or re-configuration of the mail server is not needed, allowing regular users to install the program under their home directories.

ASK was developed in Python [7], an easy to read and portable language that has been gaining a lot of popularity lately. Python is available for most Unix variants, making ASK portable across many platforms.

ASK works by reading emails from the standard input. After processing, emails can be directly stored into the user's mailbox or sent to the standard output for post-processing by other mail filters. This makes the program compatible with a number of Mail Transfer Agents and Mail Filters, like Sendmail [6], Qmail [4], Exim [12], Postfix [21] and others. Support for Procmail [19] is also embedded in the program, as well as direct delivery to "mbox" and "Maildir" style mailboxes.

Emails pending confirmation (those for which no confirmation return has been received) are stored as individual text files. The file names contain the MD5 hash sent in the confirmation, minimizing CPU utilization when matching ordinary confirmation returns. The pending mail queue can also optionally follow the Maildir format, allowing Qmail users to access and manage their queues remotely via IMAP.

ASK is normally invoked by the user's `~/ .forward` file mechanism or by procmail. Configuration is stored in the `~/ .askrc` file and all the control files and spool directories are created by default under the `~/ .ask` directory.

A flowchart of the program's operation can be seen on Figure 1. In the following paragraphs, numbers enclosed in squares represent references to the corresponding boxes in the flowchart.

Incoming mails are always checked against the whitelist [1], blacklist [2], and ignorelist [3] (in this order). The lists are implemented as text files containing a set of regular expressions. Emails can be authenticated based on the sender's email address, the recipient's email address or the message subject. Plain substring comparison is also available for simple matches.

A match in the whitelist will cause instant delivery of the email [16]. If a match is not found in the whitelist, the program tries to match the email on the blacklist and then on the ignorelist. In either case, the original email is discarded [19], but matching the blacklist means a warning message will be sent back to the sender indicating that further emails are blocked and ignored [11].

The next step is to check for *mail bounces*, or error messages sent by other MTAs [4]. Special processing takes place to prevent the delivery of every bounce sent to an invalid sender. This is discussed in more detail in Section 3.1.

ASK provides a remote queue and list management control by email. To use this feature, users must send themselves an email containing certain commands in the "Subject" header. ASK checks for *remote commands* [5] and executes the commands if appropriate. The complete set of remote commands with examples is discussed in Section 3.2.

The next step is to check for confirmation returns [6].

These messages contain a specific string in the "Subject" field, followed by the MD5 hash that uniquely identifies the message inside the pending queue. The MD5 hash is extracted from the incoming mail and used to form a file name that contains the original message. If such a file-name exists [14], the sender's email in the confirmation message *and* the one in the queued message are added to the whitelist [17]. The original message is removed from the queue and delivered [18].

If ASK detects an invalid confirmation (for which no files exist in the pending queue) the message will be labeled as "Invalid Confirmation" and delivered. This measure prevents loops with other ASK users.

A special case that deserves attention is that of spam messages with the sender's address forged to be the same as the recipient. In this case, a confirmation message cannot be sent as it would in turn be sent to the ASK user. To avoid this, ASK implements the concept of a *mailkey*: a string or short phrase that must be present in every outgoing mail sent by the ASK user. Common choices are words or short phrases from the user's signature. ASK will first check for the presence of the mailkey in incoming mails [7]. If it is found, the email is immediately delivered [16] and processing ends. If not, ASK compares the sender's address to the ASK user's address (configured at installation time) [8]. The message will be queued with a status of "Junk" if a match is found [15].

The mailkey serves a second purpose: to minimize the number of confirmations sent to replies. Ordinarily, ASK has no means of knowing if a certain message is a reply to an email sent by the ASK user or not. Most MUAs, however, quote the entire original message in a reply. This gives ASK an opportunity to detect the mailkey in replies and deliver the message without a confirmation. ASK can also be configured to automatically add the sender's email to the whitelist if a message contains the mailkey.

Sending confirmation messages to mailing-lists would be undesirable. For this reason, ASK tries to determine if a message came from a mailing-list [9] before a confirmation is sent. Mailing-list messages will be immediately queued [15] and no confirmation will be generated. The set of heuristics used to detect mailing-list and other machine-generated emails is described in Section 3.3.

At this point [10] the message has passed all the tests that could cause its delivery or dismissal:

- The message is not in the whitelist.
- The message is not in the ignorelist.
- The message is not in the blacklist.
- The message is not a bounce of any kind.
- The message is not a remote command request.
- The message is not a confirmation return.

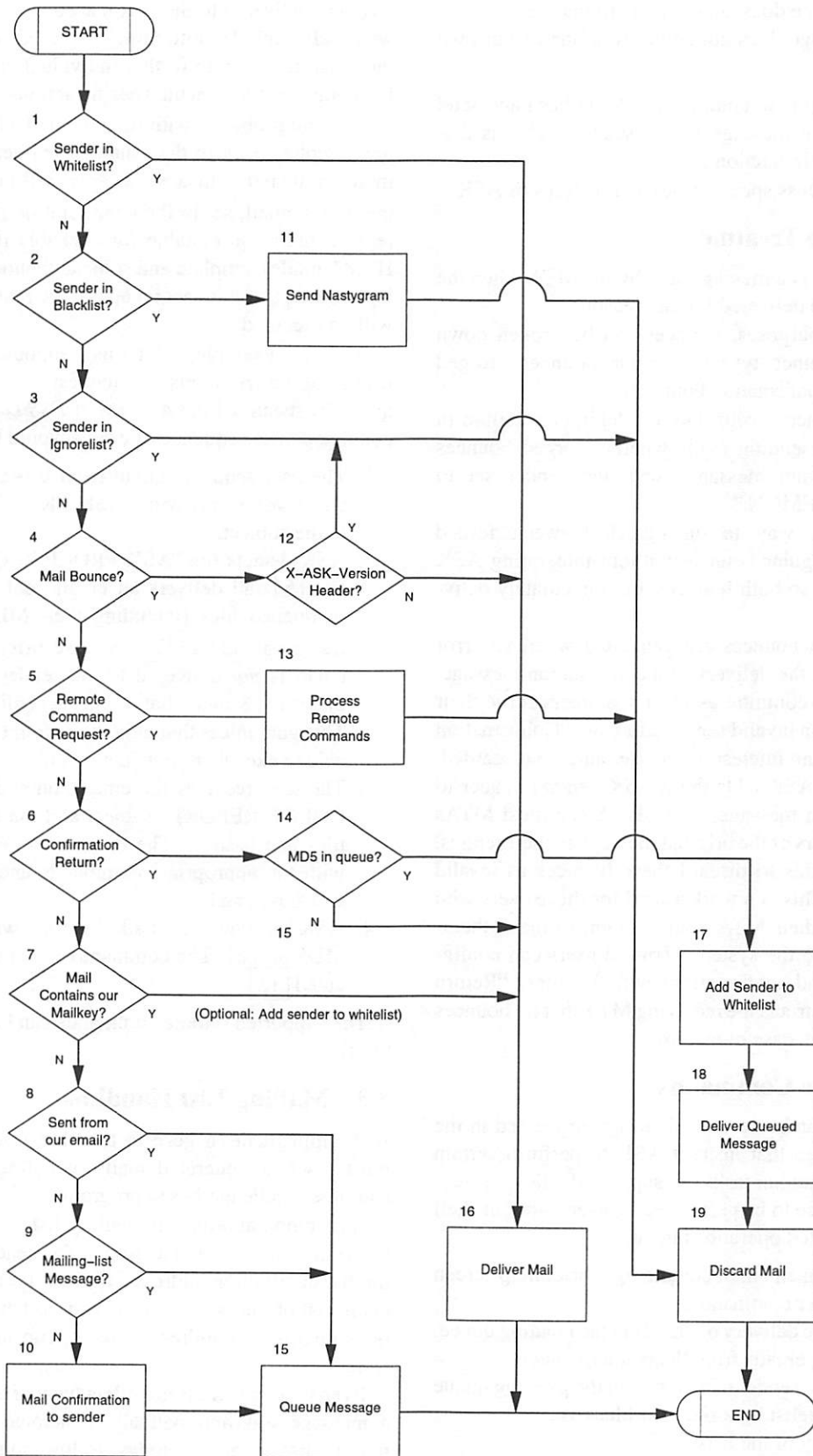


Figure 1: ASK Mail Processing Flowchart

- The message does not contain our mailkey.
- The message does not come from one of our own emails.

The final step is to compute the MD5 hash and send the confirmation message to the sender. This is discussed in detail in Section 3.4.

Next, we discuss special cases and features of ASK.

### 3.1 Bounce Treatment

A *Mail Bounce* is a message sent by the MTA when the email cannot be delivered for any reason.

For ASK's purposes, bounces can be broken down into three distinct types: regular bounces, forged bounces, and confirmation bounces.

Regular bounces occur due to a legitimate failure in the process of sending mails whereas forged bounces are actually spam messages with the sender set to "MAILER-DAEMON."

There is no way to distinguish between forged bounces and regular bounces without integrating ASK with the MTA, so both bounces are immediately delivered.

Confirmation bounces are generated when an error happens during the delivery of a confirmation message. These are very common as most spammers forge their emails to contain invalid sender addresses. Confirmation bounces are of no interest to the user and are discarded. To that effect, ASK adds the *X-ASK-Version* header to all confirmation messages it sends. Since most MTAs quote the headers of the original message in their replies, ASK can use this to discard these bounces as invalid senders [12]. This is a workaround for those users who cannot access their MTA configuration to make themselves trusted to the system. Trusted users can configure ASK to send confirmations with an empty "Return Path" which instructs the receiving MTA that no bounces should be sent in case of an error.

### 3.2 Remote Commands

Remote commands are special strings embedded in the "Subject" header that instruct ASK to perform certain operations. Common tasks are supported, allowing regular maintenance to be performed by users without shell access. Supported operations include:

- Requesting an email containing a brief help screen on the other commands.
- Forcing the delivery of emails in the pending queue.
- Removing emails from the pending queue.
- Adding the sender of an email in the pending queue to the whitelist, ignorelist, or blacklist.
- Editing any of the lists.

ASK offers two flavors of remote commands: *text mode*, which causes an editable template to be gener-

ated and delivered to the user's account or *HTML mode* where clickable "mailto" links are used in the body of the email to generate further individual emails containing commands to execute specific actions.

To avoid problems with forged email addresses, ASK never replies back to the sender when dealing with remote commands. Instead, a reply will be delivered to the user's email, set in the configuration file [13]. This reply contains an editable (or clickable if operating in HTML mode) template and some authentication tokens. Upon receipt of this second email, the requested actions will be executed.

As an example, let us suppose that user *user@domain* wants to request a listing of the queue by means of the *ASK PROCESS QUEUE* remote command. The sequence of events would be:

1. The user sends an email from *user@domain* to *user@domain* with "ASK PROCESS QUEUE" in the subject.
2. ASK detects the "ASK PROCESS QUEUE" subject [5] and delivers an email containing the list of queued files (including their MD5 hashes) to *user@domain* [12]. As a security measure, the email is *not* delivered to the sender but rather to the owner's email that was set at configuration time. This guarantees that only the account owner will be able to execute remote commands.
3. The user receives the email containing the "ASK QUEUE REPORT" subject and the list of queued files and hashes. The email is edited by the user with the appropriate commands and sent back to *user@domain*.
4. ASK receives the email, this time with the correct MD5 hashes. The commands in the email are executed [12].

The supported remote commands can be found in Table 1.

### 3.3 Mailing-List Handling

ASK implements a generic test that is able to match most machine-generated mails including mailing-lists and other challenge-based programs.

A common approach in mailing-lists is to rewrite the sender's address or add a "Reply-To" header pointing to the list distribution address. Without special treatment, confirmation messages would end up being sent to the list's distribution address, causing considerable confusion.

Even though there is no official way of telling whether a message was automatically generated or not, most mailing-list managers today follow some guidelines. ASK will not send a confirmation to a message if at least one of the following criteria is met:



Command	Description
<i>ASK HELP</i>	Produces a list of available commands.
<i>ASK PROCESS QUEUE</i>	Process the messages in the queue. Individual messages can be delivered or erased. It is also possible to directly add the sender of a given message to any of the lists (white/ignore/black).
<i>ASK EDIT WHITELIST</i>	Sends back the contents of the whitelist to be edited by the user.
<i>ASK EDIT IGNORELIST</i>	Sends back the contents of the ignorelist to be edited by the user.
<i>ASK EDIT BLACKLIST</i>	Sends back the contents of the blacklist to be edited by the user.

Table 1: Supported Remote Commands

- If the message contains at least one of the following headers: “Mailing-List,” “List-Id,” “List-Help,” or “List-Post.”
- If the message contains the “Precedence” header set to “bulk” or “list.”
- If the message contains the “Return-Path” header set to null. This header is used to indicate where error messages should be returned. If set to null, no error messages should be returned [8].
- If the username part of the sender’s email contain one of the following words: “majordomo,” “listserv,” “listproc,” “netserv,” “owner,” “bounce,” “mmgr,” “autoanswer,” “noreply,” or “nobody.” Those are very common names in mailing-lists and auto-responder return addresses.

Messages that match one of the conditions above and are not in the whitelist will still remain in the pending queue, where they can be manipulated with remote commands. No confirmation message will be sent though.

### 3.4 Sending the Confirmation Message

Sending the confirmation is the last step when processing a message from an unknown sender. Before the confirmation is sent, ASK performs two final steps in order to avoid mail loops with badly configured auto-responders and other auto-reply services:

1. ASK verifies if the current email is already queued by means of the MD5 hash (same hash, same email). This offers a basic degree of protection against services that send exactly the same message multiple times.
2. A circular list is kept with the last  $N$  addresses used when sending out a confirmation message. If the current sender’s email address appears more than  $X$  times in the list, no confirmation is sent. This guarantees that no more than  $X$  messages will ever be sent to the same sender in a given period of time. Incoming mails that generate confirmations will push the old ones out of the list, giving the sender a chance for more confirmations after some time. The values of both  $X$  and  $N$  are user configurable.

The confirmation message itself is a simple ASCII text email containing brief instructions to the sender and the MD5 hash in the “Subject” field. The confirmation has to be brief and simple or else some senders might just skip it altogether.

A typical confirmation message is presented on Figure 2.

The MD5 hash is generated by concatenating the message text to a user configurable secret, making it impossible for a spammer to get added to the whitelist by crafting a fake confirmation.

The text is easily configurable by the user and more than one language may be present in the confirmation at the same time (normally English and the user’s mother language). Ready to use templates are available in English, Spanish, French, German, Brazilian Portuguese, Dutch, Italian, and Finnish.

## 4 Evaluation

In this section we compare the effectiveness of ASK and other popular anti-spam alternatives.

### 4.1 Filtering Effectiveness

We selected five sets, from a total of 7000 emails, to evaluate the tools. We used sets *spam1* and *non-spam1*, containing 1000 messages each, to test the content-filtering and RBL tools. We used sets *spam2* and *non-spam2*, containing 2000 messages each, to train the bayesian classifier. We used a larger sample during the training phase as it improved the effectiveness of Bogofilter considerably. We used set *spam3* to test the effectiveness of a challenge-based approach.

Sets *spam1*, *non-spam1*, *spam2*, and *non-spam2* were donated by ASK users. We collected set *spam3* from SpamArchive [2]. Set *spam3* contains no messages previously queued by ASK as those are by definition unconfirmed.

Table 2 compares the effectiveness of different tools when dealing with spam.  $nFN$  represents the number of false negatives (known spam classified as a valid mail).  $nFP$  represents the number of false positives (valid mail classified as spam).  $\%FN$  and  $\%FP$  represent the per-

From: "Marco Paganini" <paganini@paganini.net>  
 Date: 23 Feb 2003 16:24:45 -0000  
 To: evil\_spammer@example.com  
 Subject: Please confirm (conf#bbdff2f4fded51313511b83120a7b37e)

<< IMPORTANT INFORMATION! >>

This is an automated message.

The message you sent (attached below) requires confirmation before it can be delivered. To confirm that you sent the message below, just hit the "Reply" button and send this message back (you don't need to edit anything). Once this is done, no more confirmations will be necessary.

This email account is protected by:  
 Active Spam Killer (ASK) - (C) 2001-2003 by Marco Paganini  
 For more information visit <http://www.paganini.net/ask>

--- Original Message Follows ---

Date: Sun, 23 Feb 2003 08:24:34 -0800 (PST)  
 From: evil\_spammer@example.com  
 Subject: SPAM TEST!  
 To: paganini@paganini.net

Hello.  
 This could be spam.

Figure 2: A Typical Confirmation Message

centage of false negatives and false positives, respectively.

We distributed RBL providers in three groups, named "RBL Group 1," "RBL Group 2," and "RBL Group 3." The first group contains only one RBL provider and represents the minimum protection case. The second group contains a more reasonable mix with three distinct RBL providers. Group 3 represents an extreme case with nine RBL providers. The composition of these groups is listed in Appendix A.

We tested DCC using two different thresholds (number of reports for a message to be considered spam). In the first test, we considered one report enough to mark the message as spam. The second test used a more conservative approach where five reports are needed as opposed to one.

Program	nFN	nFP	%FN	%FP
Bogofilter	22	16	2.2	1.6
DCC (Threshold 1)	799	0	79.9	0.0
DCC (Threshold 5)	925	0	92.5	0.0
RBL Group 1	997	10	99.7	1.0
RBL Group 2	652	16	65.2	1.6
RBL Group 3	420	308	42.0	30.8
SpamAssassin	118	20	11.8	2.0

Table 2: Compared Effectiveness of anti-spam tools

Bogofilter proved to be the most effective content-filtering tool in its category, followed by SpamAssassin. The percentage of false positives is very close for these

two tools. DCC had a considerable percentage of false negatives but presents no false positives.

Using one RBL proved to be almost 100% ineffective in our tests, with only three spam emails detected correctly. Raising the number of RBLs checked to three resulted in a 65.2% rate of false negatives, but also raised the percentage of false positives to 1.6%. Group 3 represents an extreme case, with nine RBLs being checked. Even under these circumstances, a high mark of 42% false negatives was verified, with a non-viable mark of 30.8% false positives. Of all solutions tested, RBLs were visibly the worst performers.

#### 4.1.1 ASK Filtering Effectiveness

To test ASK's effectiveness, we sent 1000 confirmations with a specially crafted envelope address and sender (to catch bounces and replies). We waited for one week for confirmations to return. The results can be seen on Table 3.

Description	Number of messages
Invalid Address (Bounced)	637
No response	259
Malformed emails	86
Responses received	18
<b>Total</b>	<b>1000</b>

Table 3: ASK Effectiveness Test

Of the 18 responses received, 14 were from one specific online marketing company and obviously unsolicited. The remaining four came from bargain notifica-

tion services, apparently sent after the user subscribed to their services. Only three replies kept the original confirmation code in the message subject (without which, no message delivery takes place), bringing the rate of false negatives to 0.3%. Even if we assume that all spammers could keep the "Subject" header intact, we will still have a rate of false negatives of only 1.8%.

False positives in a challenge-authentication system occur when valid senders do not reply to confirmation messages. Systematic determination of the false positive rate in a challenge-based authentication tool is a difficult task. A confirmation message would have to be re-sent to known valid senders, causing all sorts of problems. Also, some senders who replied to the first confirmation could get confused by the second confirmation request and ignore it completely.

In an effort to understand other users' experiences with spam and how ASK performs in their environments, we submitted a simple survey [1] to the main ASK mailing-list during March of 2003. A total of 32 users responded. The results can be seen in Table 4.

The results indicate that most users have been using ASK from 6 months to one year, with a substantial amount of new users in the last three months.

Most ASK users received 10 to 20 spam messages a day (33%), with a sizable part receiving 20 to 50 (28%) spam messages a day. It is interesting to note that the distribution is fairly even across the categories, indicating that ASK caters to all classes of users when it comes to amount of spam received.

After beginning to use ASK, 61% of the users report that practically no spam is present, and 21% report that only 1 to 5 spam messages are received per month. This brings the total users with a substantial reduction of spam to 82%.

One important aspect of the survey is to verify whether spammers reply to confirmation messages or not. 35% reported no cases of spam delivered due to this reason while 28% reported one or two cases. This brings the total percentage of users with less than three cases of spammers replying to confirmations to 63%. Another 28% reported between two and ten cases since ASK was installed, which still can be considered a good mark considering that most users seem to be using ASK for more than six months.

Another point we tried to clarify is the possibility of valid users not responding to the confirmations (false positives). Most users (42%) reported very few cases of false positives (1 to 2). 31% reported no cases at all. These results combined indicate that 73% of those who responded to the survey had no significant problems with false positives. This seems to indicate that, as a rule, valid senders tend to reply to confirmation messages.

## 4.2 Compatibility

ASK requires a Unix/Linux compatible operating system. ASK was written in Python and should run with no or few modifications under most Unix variants.

ASK is compatible with most major MTAs available today, including Sendmail, Qmail, Postfix, and Exim. Procmail support is also available. Any MTA capable of delivering to a pipe should work without problems. ASK natively supports mbox and Maildir mailbox formats, with MH planned.

Incoming mails are read on the standard input, making it possible to cascade ASK with other anti-spam solutions like SpamAssassin or a multitude of Bayesian filters and RBLs.

## 4.3 Performance

To evaluate ASK's performance, we selected a set of 1000 spam messages contributed by ASK users, totaling 14MB of data. On the average, each message has 14KB and 457 lines of text. Text lines have an average length of 31 bytes.

We configured ASK to send a confirmation to each message. ASK took 524 seconds (on a Pentium III/500MHz workstation) to process all messages, bringing the average processing overhead to 0.52 seconds per email received.

During this test, we tried different logging levels (0, 1, 10) and different list sizes (5, 100, 500 lines). No significant variation was detected in the results.

To measure the network overhead, we configured ASK to keep the first 50 lines of each message in the confirmation. This resulted in an average size of 3.3KB per confirmation message, with two languages selected, totaling 3.3MB of extra network traffic. We estimate the overhead of confirmation bounces to be around 2MB, but it can be avoided by configuring the system to send confirmation messages with a null "Return-Path."

Another area of interest is the overhead in processing remote commands. ASK took 30 seconds to process a queue with 1000 messages. This is the most expensive remote command, as every file in the queue has to be opened and investigated. Proper queue maintenance should keep the number of queued files well under 1000, reducing considerably the time for this operation.

The CPU overhead is acceptable for most sites, but something to be considered for high volume servers. The network traffic overhead should be of no concern unless network service is paid by volume. If this is the case, a few bandwidth reduction measures can be taken, such as reducing the number of lines quoted from the original message in the confirmation or limiting confirmation messages to only one language.

Question	Answers	Percentage
Total time using ASK	Less than a month	12%
	1 to 3 months	28%
	3 to 6 months	9%
	6 months to one year	42%
	More than one year	9%
Number of spam messages received per day before ASK	1 to 10	33%
	11 to 20	18%
	20 to 50	28%
	More Than 50	21%
Number of spam messages received per month after ASK	Practically none	61%
	1 to 5	21%
	Between 5 and 10	9%
	More than 10	9%
Number of times a spammer replied to the confirmation	0	35%
	1 to 2	28%
	2 to 10	28%
	10 to 20	6%
	More than 20	3%
Number of times a valid sender failed to reply to a confirmation	0	31%
	1 to 2	42%
	2 to 10	15%
	10 to 20	6%
	More than 20	6%

Table 4: ASK User Survey Results

## 5 Related Work

In this section we discuss other challenge-based authentications and compare some of their key features to ASK.

### 5.1 Tagged Message Delivery Agent

Tagged Message Delivery Agent (TMDA) is a spam-reduction solution by Jason R. Mastaler. TMDA is also written in Python and integrates well with most MTAs and MUAs.

One of the key problems in challenge-authentication tools is how to match the confirmation return to the stored message, as very few headers from the original message are kept in the reply. ASK sends the MD5 hash that identifies the message (the cookie) in the “Subject” header, knowing that most MUAs quote the original subject in replies.

TMDA takes another approach to this problem. Instead of relying on the contents of the “Subject” header, the sender’s own email is rewritten to contain the cookie in it. This is possible due to the use of *extension addresses*, a configurable MTA feature that allows one account to receive emails with differentiated recipient addresses. Under this scheme, emails going to `foo@bar` and `foo-extension@bar` will both be delivered to user `foo` at the host `bar`. TMDA uses the “-extension” part to include all sorts of control messages

and cookies that need to be present in the reply.

A typical message exchange between a TMDA user called `tmlda@domain` and a non-TMDA user called `user@domain` is something as follows:

1. User `user@domain` sends `tmlda@domain` a message.
2. TMDA intercepts the message for `tmlda@domain` and generates a confirmation to `user@domain`. The confirmation’s sender address is rewritten to something like `tmlda-confirm-cookie@domain` where the cookie is an alphanumeric sequence that identifies the original message.
3. The user sees the confirmation and replies. The reply is sent to `tmlda-confirm-cookie@domain`.
4. TMDA receives the confirmation return and extracts the cookie from the recipient’s email address. The message is dequeued and delivered.

TMDA also presents the concept of *tagged messages* (hence its name): outgoing emails may have the sender’s address rewritten to contain special tags that bind that email to that recipient (future emails from that recipient will only be accepted on the special, tagged email), dated addresses (emails that are valid only throughout a date



range) and keyword addresses, temporary addresses that work indefinitely until manually revoked (for Web-based services).

The utilization of extension addresses adds robustness to the system and the possibility of tagged messages. However, it requires complete control over the MTA configuration files at the ISP level (or an ISP with extension addresses enabled, a rather uncommon feature). Furthermore, some integration issues exist with Sendmail, which does not provide envelope information to programs invoked from the user's `~/forward` file. In these cases, procmail must be configured as the *local mailer* in order to make all the required information available to TMDA.

## 5.2 Qconfirm

Qconfirm is a challenge-authentication system written in the C Language by Gerrit Pape for Qmail users.

Unlike ASK, Qconfirm is not a single program but rather a group of smaller programs with specific tasks (following Qmail's style). Incoming messages are checked by the `qconfirm-check` program which is also responsible for sending confirmation messages to unknown senders.

Pending messages are held in the qmail queue. `qconfirm-check` will send confirmation messages with an empty envelope sender and the "From" header set to "user-qconfirm-key@host," and create a file named `.qmail-qconfirm-key` under the user's home directory.

Confirmation returns have the recipient set to `user-qconfirm-key@host`. This will trigger the `.qmail-confirm-key` file previously created and invoke `qconfirm-accept` to deliver the queued message and add the sender's address to the `.qconfirm/ok` directory (Qconfirm's whitelist).

Common queue and list maintenance tasks are performed by means of a command line utility called `qconfirm`. This program can also be executed remotely by creating special Qmail control files to invoke `qconfirm-control`. This provides an interface similar to ASK's remote commands.

Qconfirm is lightweight and fast but relies heavily on the infrastructure offered by Qmail. This leaves Qconfirm as an option for Qmail users only.

## 6 Conclusions

We presented ASK, a challenge-authentication system that authenticates senders before their emails are delivered. In our tests with 1000 spam messages, ASK was able to block 99.7% of all spam messages, meaning that only 3 spam messages got through.

## 6.1 Future Work

Currently, ASK is not directly integrated with the MUA or the MTA, meaning that it has no knowledge about outgoing emails sent directly by the user. This design decision was taken to allow users without supervisory rights to install and use the program. Unfortunately, it creates certain situations that could be used by spammers like sending emails that appear to be confirmation messages from other ASK users or forging email bounces. MTA and MUA integration will be offered by means of an *SMTP proxy agent* and an *MTA wrapper*. Both approaches offer ASK the opportunity to pre-process outgoing messages before the actual delivery takes place. Extension addresses can be used to rewrite the outgoing envelope address so that invalid confirmation returns and MTA bounces can be correctly tracked.

The problem of emails coming from unknown sources will be addressed with the introduction of two new concepts: *Bounded addresses* and *user confirmation mode*. Bounded addresses create a temporary address that whitelists the first sender who sends an email to it. That sender will be forever tied to that particular email. This is similar in concept to TMDA's "Keyword addresses," with the difference that they become bound to one particular sender after the first use. This creates a "throw-away" email address that can easily be revoked in case of abuse.

User confirmation mode will be available to those who cannot change their MTA configurations or do not desire to make use of extension addresses. Under this mode, confirmation messages are sent to the account owner instead of the sender. This allows the owner to perform a reply and whitelist an email coming from an unknown account. Once the first reply is received, ASK can resume the normal mode of operation.

Other smaller features are also planned, like MH style mailbox support, automatic queue cleanup, and augmented pattern matching for the lists, including full header and body regular expression matching and boolean NOT qualifiers among others.

## 6.2 Availability

ASK is Open Source Software released under the GNU GPL Software License. The program's home page, including download and documentation links is located at [www.paganini.net/ask](http://www.paganini.net/ask)

## 7 Acknowledgments

We would like to thank Daniel Bastos, Durval Menezes, Charles P. Wright and all others who contributed with their mailboxes, making the effectiveness test possible. We thank Jason Mastaler and Gerrit Pape for their help reviewing the "Related Work" Section. Thanks go to

the anonymous Usenix reviewers and specially our shepherd, Erez Zadok, for his guidance and support.

## A Realtime Blackhole Lists

Here we list the RBL providers used for the tests performed on Section 4.1.

- Group 1:
  1. Open Relay Database (ORDB), [www.ordb.org](http://www.ordb.org)
- Group 2:
  1. Spamhaus, [www.spamhaus.org](http://www.spamhaus.org)
  2. Blitzed Open Proxy Monitor List, <http://opm.blitzed.org>
  3. Relays.Osirusoft.com, <http://relays.osirusoft.com>
- Group 3:
  1. Spamhaus, [www.spamhaus.org](http://www.spamhaus.org)
  2. Blitzed Open Proxy Monitor List, <http://opm.blitzed.org>
  3. Relays.Osirusoft.com, <http://relays.osirusoft.com>
  4. Open Relay Database (ORDB), [www.ordb.org](http://www.ordb.org)
  5. Not Just Another Bogus List, <http://dnsbl.njabl.org/>
  6. Extreme Spam Blocking List (xbl), <http://xbl.selwerd.cx/>
  7. Fiveten, [www.five-ten-sg.com/blackhole.php](http://www.five-ten-sg.com/blackhole.php)
  8. Spamcop Blocking List, <http://spamcop.net/bl.shtml>
  9. Distributed Server Boycott List (DSBL), <http://dsbl.org/main>

## References

- [1] Ask user survey. [www.paganini.net/ask/cgi-bin/survey.cgi](http://www.paganini.net/ask/cgi-bin/survey.cgi), March 2003.
- [2] Spamarchive. [www.spamarchive.org](http://www.spamarchive.org), 2003.
- [3] SohoAny Associates. Dynamicmailer. [www.sohoany.com/Dynamicmailer.html](http://www.sohoany.com/Dynamicmailer.html), 2003.
- [4] Dan J Bernstein. Qmail. [www.qmail.org](http://www.qmail.org), 2003.
- [5] Federal Trade Commission. Email address harvesting: How spammers reap what you sow. [www.ftc.gov/bcp/online/pubs/alerts/spamalert.pdf](http://www.ftc.gov/bcp/online/pubs/alerts/spamalert.pdf), November 2002.
- [6] The Sendmail Consortium. Sendmail. [www.sendmail.org](http://www.sendmail.org), 2003.
- [7] The Python Software Foundation. Python. [www.python.org](http://www.python.org), 2003.
- [8] J. Klensin. Simple mail transfer protocol. Technical Report RFC 2821, AT&T Laboratories, April 2001.
- [9] Mail Abuse Prevention System LLC. Mail abuse prevention system rbl. <http://mail-abuse.org/rbl/>, 2003.
- [10] Sharon Machlis. Spam's getting more sophisticated. [www.computerworld.com/softwaretopics/software/groupware/story/0,10801,7%7704,00.html](http://www.computerworld.com/softwaretopics/software/groupware/story/0,10801,7%7704,00.html), January 2003.
- [11] Jason R. Mastaler. Tagged message delivery agent. <http://tmlda.net>, 2003.
- [12] University of Cambridge. Exim. [www.exim.org](http://www.exim.org), 2003.
- [13] Gerrit Pape. Qconfirm - request delivery confirmation for mail. <http://smarden.org/qconfirm/>, 2003.
- [14] Vipul Ved Prakash. Vipul's razor. <http://razor.sourceforge.net/>, 2003.
- [15] Eric S Raymond. Bogofilter. <http://bogofilter.sourceforge.net/>, 2003.
- [16] R. Rivest. Simple mail transfer protocol. Technical Report RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992.
- [17] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.
- [18] Rhyolite Software. Dcc. [www.rhyolite.com/anti-spam/dcc/](http://www.rhyolite.com/anti-spam/dcc/), 2003.
- [19] Philip Guenther Stephen R. van den Berg. Procmail. [www.procmail.org](http://www.procmail.org), 2003.
- [20] The SpamAssassin Development Team. Spamassassin. [www.spamassassin.org](http://www.spamassassin.org), 2003.
- [21] Wietse Venema. Postfix. [www.postfix.org](http://www.postfix.org), 2003.

# Learning Spam: Simple Techniques For Freely-Available Software

Bart Massey Mick Thomure  
Raya Budrevich Scott Long  
Computer Science Department  
Portland State University  
Portland, OR USA

{bart, thomure, binargrl, scottl}@cs.pdx.edu  
<http://oss.cs.pdx.edu/psam>

## Abstract

The problem of automatically filtering out spam e-mail using a classifier based on machine learning methods is of great recent interest. This paper gives an introduction to machine learning methods for spam filtering, reviewing some of the relevant ideas and work in the open source community. An overview of several feature detection and machine learning techniques for spam filtering is given. The authors' freely-available implementations of these techniques are discussed. The techniques' performance on several different corpora are evaluated. Finally, some conclusions are drawn about the state of the art and about fruitful directions for spam filtering for freely-available UNIX software practitioners.

## 1 Introduction

There has been a great deal of interest of late in the problem of automatically detecting and filtering out unsolicited commercial e-mail messages, commonly referred to as *spam*. (While the Hormel Corporation, owners of the "Spam" trademark, are not happy about the choice of name, they have acceded to the popular usage. For a further etymology see [14].) Recent dramatic increases in spam volume have combined with the success of a number of new filtering methods to make automated spam filtering highly successful. The SpamAssassin [20] mail-filtering tool is one such tool. SpamAssassin uses a large manually-generated feature set and a simple perceptron classifier with hand-tuned weights to select *ham* (non-spam) messages and discard spam.

Much current interest has focused around the role of *machine learning* [5, 15] in spam filtering methodologies. This paper describes the basics of machine learning and several simple supervised machine-learning algorithms that are effective in filtering spam. The authors have made implementations of these algorithms publicly available, along with various kinds of feature data from several corpora used to evaluate the algorithms. These implementations and data are used to help eval-

uate the relative merits of these algorithms, and suggest directions for future work.

## 2 Context

The spam problem has received increasing attention in recent years. As a result, a number of approaches for dealing with the problem have been proposed. A recent issue of Wired magazine [11] lists a variety of popular strategies. *Blacklists* such as spamcop.net attempt to stop spam by preventing spam-delivering hosts from communicating with the rest of the Internet, or at least with the victim machine. *Distributed identification systems* such as Vipul's Razor allow users to manually identify spam for collaborative filtering. *Header analysis* can be used to eliminate messages that have malformed or unusual headers or header fields, as well as messages that have invalid return addresses or sender information. *Legal approaches* are gaining currency at both the U.S. State and Federal levels, including proposed penalties for unsolicited commercial e-mail and anonymous commercial messages. (It should be noted that the legal approach is widely credited with largely eliminating unsolicited commercial messages via FAX.) The range of approaches is growing rapidly in response to the increase in spam traffic: approaches not noted by Wired include *whitelist* systems such as Active Spam Killer [16], a mailback system that attempts to verify that e-mail is ham by requiring a confirmation message from unknown senders.

A recent conference on the spam problem at MIT [8] was nearly overwhelmed by the volume of attendees. The presentations were quite productive; many critical points were raised about spam filtering that deserve wider attention by the open source community. There was far too much useful information to summarize here: perhaps one example will suffice. There seems to be a widespread perception that false positives (ham messages flagged as spam by filters) are "intolerable" in spam filtering.



The MIT Spam Conference presenters mentioned several reasons why insisting on zero tolerance for false positives can lead users to wrongly reject spam filtering as a technology. As many researchers have noted, the absolute prohibition of false positives can only be justified by assuming that they have infinite cost: while a false positive may have a cost much larger than a false negative [10], this cost is not infinite. Further, false positive rates of most filtering algorithms can be lowered in a tradeoff for false negative rates. Finally, a good spam filter may actually exhibit super-human classification performance: after all, this is the sort of repetitive and error-prone task that a human may be expected to perform poorly [12]. The unsophisticated filters reported here uniformly achieve false positive rates of just a few percent: the authors informally estimate their human false positive rates to be in a similar range. Third, the false positive rate of a single spam filter is somewhat irrelevant: both ensembles of filters and the combination of filtering with other approaches to spam detection can largely take care of the overall false positive problem. Finally, spam will only be sent if it is profitable: in the long haul, widespread use of filters may change the economics of spamming enough to largely eliminate the problem [7].

Research in spam filtering within the freely available software community is currently proceeding quite quickly: some of what is said here about the state of the art will no longer be true by the time it is published. The general system engineering and machine learning principles that are key to the spam elimination effort, however, should still be valuable for some time.

### 3 Machine Learning

Machine learning is a field with a broad and deep history. In general, a machine learner is a program or device that modifies current behavior by taking into account remembered past results. This is a broad definition. However, much of machine learning research is focused on *inductive learning*, in which general rules are built based on a *corpus*, a set of specific examples. In spam filtering (and many other applications) the corpus consists of pre-classified examples, and the learned rules are used to classify e-mail as either ham or spam.

This paper gives greatest emphasis to *supervised learning*. In supervised learning, the examples to be used for learning are collected and processed during a *training phase*. The learned rules are then used without further modification during a *classification phase*. *Reinforcement learning*—on-line correction of the learned rules in response to classification errors—is also quite valuable. This allows the system to adapt to changing conditions, such as user preferences or spam content. The simplistic approach of re-learning the entire corpus,

including newly acquired classifications, can suffice if the learner is sufficiently fast on large inputs.

Supervised machine learning for spam classification begins with a corpus consisting of a collection of correctly classified ham and spam messages. In the *feature selection* stage, key features of the corpus are identified that distinguish ham from spam. In the *training* stage, the selected features of the corpus are studied to learn characteristics that differentiate spam from ham messages. Concurrently or subsequently, a *validation* stage is often used to check the accuracy of the learned characteristics. Finally, the learned knowledge is used in a *classification* stage that filters spam by giving a classification to each *target* message in the classification set.

Some important considerations in supervised learning involve management of the corpus. For accuracy's sake, one would like to use the entire corpus as training data. Unfortunately, this makes validation quite difficult: the classifier will appear to perform unrealistically well when asked to classify the messages on which it was trained. Fortunately, in most problem domains the number of training instances needed to learn with a given accuracy grows only logarithmically with the size of the *hypothesis space*, the set of concepts that must be distinguished. Thus, it is customary to split the corpus into a *training set* and a *validation set*. A rule of thumb in machine learning is to make the validation set consist of a randomly selected third of the corpus. There are much more sophisticated methods for validation that improve on the quality of this approach, but the simple method will suffice for most cases.

It is reasonable to be concerned about the minimum corpus size required for full accuracy. As corpus size increases, machine learning algorithms tend to asymptotically approach their maximum accuracy. Figure 1 shows the accuracy of a number of different machine learning algorithms on increasingly large subsets of a synthetic corpus discussed in Section 7. The figure shows that for the algorithms discussed here, 100–1000 messages are sufficient to achieve maximal accuracy. The variance at low corpus sizes is due to statistical error, and represents a large inter-run variance.

A risk that must be countered when training a machine learner is *overtraining*: building a learner that classifies based on quirks of the training set rather than general properties of the corpus. Figure 2 shows the change in classification rate during training for the perceptron of Section 4.2.4 on the 15,000 instance personal e-mail corpus described in Section 7. In the figure, the accuracy of classification on the training set continues to increase, while the accuracy on the validation set actually begins to drop. This explains the need for an independent validation set: training should stop when maximal validation set accuracy is reached.



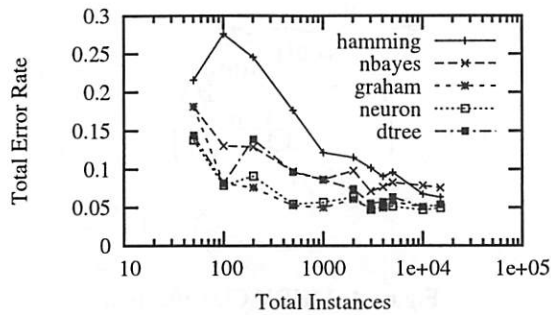


Figure 1: Learning Rate

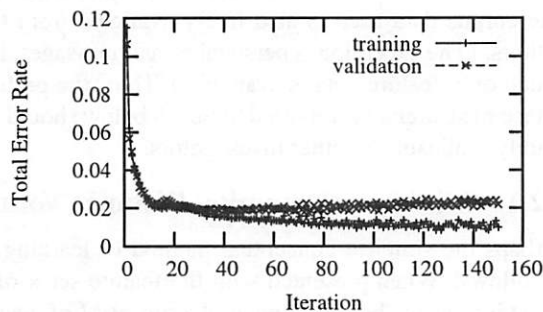


Figure 2: Overtraining

An important tradeoff in spam filtering is between false positive (ham messages flagged as spam) and false negative (*misses*, spam messages flagged as ham) rates. A detector that always says “ham”, after all, will never experience a false positive. In communications theory, this tradeoff is illustrated by a *receiver operating curve* that shows the tradeoff between rates. Most spam filters prefer to operate with a bias that minimizes the total error. False positives, however, are generally much more expensive than false negatives, so it may be desirable to operate the filter outside of its optimal range. Figure 3 shows receiver operating curves for several spam filters on the synthetic corpus discussed below. The spam filters were biased by varying the percentage of spam from 5% to 95%: for these filters, this caused the detection profile to shift. The strong preference of the filters for operating with a particular optimal bias is notable.

The accuracy of the corpus is also a concern. It is to be expected that a certain amount of misclassification of messages and mis-recognition of features will be present in the data. Section 7 discusses some of the characteristics of the corpora used here. While these corpora have received a great deal of attention from a variety of sources, they nonetheless seem to have some residual misclassification.

Different learners may cope with different types of features and classifications. Ultimately, spam filtering tends to concern itself with a binary classification: ham

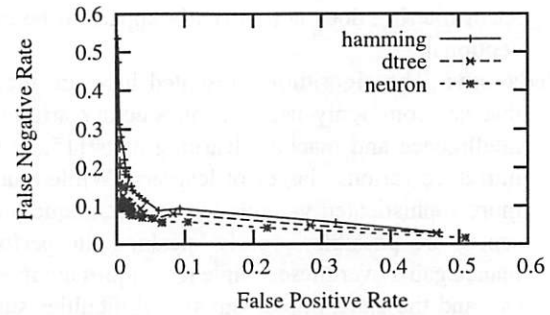


Figure 3: Receiver Operating Curves

vs. spam. More sophisticated document classifiers can provide both more detailed classification outputs (e.g. “Nigerian spam”, “message from Mom”) and more precise estimates of their classification confidence.

Some machine learners (notably neural nets) can handle continuous feature values. Many learners are restricted to discrete feature domains, and one of the algorithms discussed here is tailored to binary features. The learning algorithms described in this paper have been set up to use binary features, for several reasons. The fact that binary feature data can be handled by essentially any inductive learner permits the comparison of a wide range of approaches. The binary-feature version of a typical learning algorithm is easier to explain: the mathematical notation is complicated enough without worrying about many-valued features. Perhaps most importantly, the common types of binary feature detectors are more difficult for a spammer to manipulate. For example, if the number of occurrences of a particular feature in a given message is considered, a spammer can load a message up with repeated instances of a “good” feature and overwhelm the spam-related features of the message.

## 4 Learning Methods

There are a huge range of approaches to machine learning discussed in the literature. Several criteria have been used to select algorithms for presentation:

**Simplicity:** First and foremost, the algorithm must be comprehensible and easily implementable by UNIX developers of freely available software. Algorithms comprehensible only to machine learning experts have been eschewed: they often offer only a marginal increase in performance in any case.

**Currency:** Most of the algorithm families currently being used by freely-available spam detectors are represented in this sample. A glaring omission is genetic algorithms. The range of algorithms and implementations in this category is enormous, making it difficult to select a canonical candidate. In addition, the performance of genetic algorithms in

spam filtering does not currently appear to be exceptional.

**Pedagogy:** The algorithms presented here are those that are commonly used in introductory artificial intelligence and machine learning texts [15, 5] to introduce various classes of learners. While much more sophisticated variants of each technique presented are possible, grossly speaking the performance gains over these simple techniques are modest, and the extra implementation difficulties substantial.

It is worth emphasizing this last point again. More sophisticated variants of each of the algorithms presented here have already been applied to spam filtering. Simple algorithms tend to perform reasonably well, so focusing on them is practical. More importantly, the study of simple algorithms is intended to be inspirational, leading to further investigation by spam filtering practitioners in the freely available software community.

## 4.1 Notation

As mentioned earlier, this paper considers binary features for binary classification. A feature detector is applied to an e-mail message to produce a set  $\mathbf{x} = x_1 \dots x_m$  of binary features of the message. The binary classification  $c$  of the message may be given or may be the quantity to be determined: this classification is either  $+$  indicating spam, or  $-$  indicating ham. Both the classified feature detector output  $\mathbf{x} \rightarrow c$  and the original e-mail message are informally referred to as an *instance*. In the absence of other context the feature detector output will be implied. Negation will be represented with an overline, thus  $\bar{x}_i$  is true when feature  $x_i$  is absent.

The instances considered here are drawn from up to three disjoint sets: a set  $T$  of training instances, a set  $V$  of validation instances, and a set  $C$  of classification instances. When more than one instance is involved, additional subscripts for features and classifications will represent the instance; for example  $\mathbf{x}_j = x_{1j} \dots x_{mj}$ . The set of positive and negative instances drawn from a set  $S$  will be represented by  $S_+$  and  $S_-$  respectively. The set of instances with feature  $i$  true and false will be represented by  $S_{x_i}$  and  $S_{\bar{x}_i}$  respectively.

## 4.2 Techniques

In this section, several supervised learning techniques are considered. An extremely simple baseline algorithm is presented, intended partly to illustrate concepts and to provide a standard of comparison. A discussion of commonly-used and important algorithms ensues, concluding with a decision-tree method. Finally, an advanced approach using multilayer neural networks is discussed.

The authors have made UNIX utility implementations

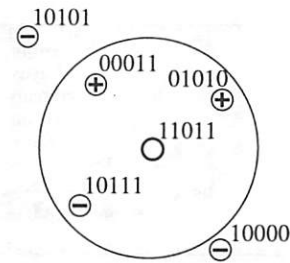


Figure 4: MHDV Classification

of each of the algorithms described in this section freely available: see **Availability** at the end of this document. The corpus data used is also freely available from the authors. (The exception is personal e-mail messages, for which only feature data is available.) Thus, the performance measurements reported in Sec. 8 below should be readily replicable by other investigators.

### 4.2.1 Minimum Hamming Distance Voting

Perhaps the simplest conceptual method of learning is as follows. When presented with the feature set  $\mathbf{x}$  of a target instance to be classified, find a subset  $M$  of the instances in the training set  $T$  with the same feature values  $\mathbf{x}$ . Then classify  $\mathbf{x} \rightarrow c$  if  $|M_+| > |M_-|$ , and  $\mathbf{x} \rightarrow \bar{c}$  otherwise. (Actually, ties should be randomized appropriately.)

This brute-force method is simple to implement, but it has drawbacks. Foremost of these is that given a reasonably small training set and reasonably large number of features, it may be unlikely to find any training instances whose features match those of the target instance. In this situation, the error rate may be very high.

Minimum Hamming Distance Voting (MHDV) is an adaptation of this “brute force” method designed to achieve higher accuracy for a given training set size. The Hamming distance  $h(\mathbf{x}, \mathbf{x}')$  between two binary vectors  $\mathbf{x}$  and  $\mathbf{x}'$  is defined to be the number of bit positions in which  $\mathbf{x}$  and  $\mathbf{x}'$  differ. Thus,  $h(\mathbf{x}, \mathbf{x}') = \#(\mathbf{x} \oplus \mathbf{x}')$ , where  $\oplus$  is the exclusive-or operator and  $\#$  is the population count or Hamming weight: the number of 1 bits in the vector.

MHDV generalizes brute force learning via the simple mechanism of using nearby instances rather than identical ones. Consider a target instance  $\mathbf{x}$  and a training set  $T$ . Let  $M$  be the subset of  $T$  with minimal Hamming distance from  $\mathbf{x}$ . A target message  $\mathbf{x}$  is classified as spam if  $|M_+| > |M_-|$  and ham otherwise. Ties are broken randomly. Formally, MHDV is an *instance-based* learning method, specifically a *k-nearest-neighbor* algorithm with  $k = 1$ . Figure 4 illustrates the MHDV classification process: the target instance (the empty dot with feature vector 11011) is classified positive in accordance with the majority of its distance-2 neighbors.

To the best of the authors' knowledge, instance based learning has not previously been proposed as a spam filtering methodology. There are serious advantages to this approach, but also serious drawbacks. The accuracy of MHDV improves dramatically as a function of the size of  $M$ , and therefore as a function of the size of  $T$ . But in a naïve implementation a single classification requires comparing the target  $x$  to each instance in  $T$ , and thus time  $o(m|T|)$  where  $m$  is the number of features. (More sophisticated implementations can use similarity hashing techniques to improve this performance somewhat.) The storage of the large set of instances is also a burden, although storage is increasingly inexpensive.

Note that reinforcement learning is easy with MHDV: simply put misclassified instances into the training set with the correct classification. MHDV should be easily extensible to discrete or continuous features that obey a distance metric.

### 4.2.2 Naïve Bayesian

An interesting class of supervised learning algorithms focuses on probabilistic interpretation of training data. One of the simplest of these is the so-called *naïve Bayesian* approach. Bayes' Rule famously notes that

$$\text{pr}(S|x) = \frac{\text{pr}(x|S) \cdot \text{pr}(S)}{\text{pr}(x)}$$

where  $\text{pr}(S|x)$  is the probability that an instance is spam given that it has the given feature set,  $\text{pr}(x|S)$  is the probability that it has the given feature set given that it is spam (an important distinction),  $\text{pr}(S)$  is the overall probability that a message is spam, and  $\text{pr}(x)$  is the probability of receiving a message containing the given features. Crucially, the quantities on the right-hand side of the equation can all be measured, under the (wrong, but surprisingly harmless in practice) assumption that the features  $x_1 \dots x_m$  are *independent*, having no particular statistical relationship. A Naïve Bayes classifier thus classifies a message as spam when

$$\frac{(\prod_{i=1}^m \text{pr}(x_i|S)) \cdot \text{pr}(S)}{\text{pr}(x)} > \frac{(\prod_{i=1}^m \text{pr}(x_i|H)) \cdot \text{pr}(H)}{\text{pr}(x)}$$

Note that the denominator is constant across the inequality and can be dropped.

Some algebraic and probabilistic manipulation yields a decision rule that classifies a message  $x$  as spam if and only if

$$|T_+| \prod_{i=1}^m \frac{|T_{+,x_i}|}{|T_+|} > |T| \prod_{i=1}^m \frac{|T_{-,x_i}|}{|T|}$$

To oversimplify, Naïve Bayes classifies an instance as spam if it shares more significantly in the features of

spam than in the features of non-spam. The rule also takes into account the *a priori* probability that the message is spam, i.e. the overall spamminess of the training set. A statistical adjustment (see [15]) is used for features that appear rarely or not at all with a given sign in the corpus. It is also common to take logarithms to turn the product computation into a sum computation: this greatly improves numerical robustness at a slight expense in performance.

The Naïve Bayes classification rule can be seen as a relative of the MHDV rule that uses the feature set in a different, more principled fashion. Another major difference is that the set sizes used in the decision rule can be computed during the learning phase, and the training data then discarded. This makes classification more efficient than with MHDV.

Naïve Bayes learning is relatively simple to implement, and accommodates discrete features reasonably well. It is not quite as accurate or robust as some other methods, but is highly efficient to train. Reinforcement learning is also easy: the relevant set sizes are simply continuously updated with newly-classified instances.

### 4.2.3 Graham

As mentioned earlier, much of the interest in Bayesian methods in the freely available software community was inspired by Graham's article *A Plan For Spam* [6]. The machine learning approach used by Graham was an informal probabilistic one: Robinson [18] later elucidated the relationship between Graham's technique and Naïve Bayesian methods.

In essence, Graham's method is similar to Naïve Bayesian: the *a priori* probabilities of a message's words are combined to yield a likelihood that the message is or is not spam. Specifically, Graham classifies a message as spam if

$$\prod_{i=1}^m \frac{\frac{|T_{+,x_i}|}{|T_+|}}{\frac{|T_{+,x_i}|}{|T_+|} + \frac{|T_{-,x_i}|}{|T|}}$$

is greater than 0.9. The features  $x_i$  used in the calculation are those words whose contribution to the product differs most from 0.5: roughly speaking, these are the high-gain words (Sec. 4.2.5). Various empirical adjustments are made to the above formula in the implementation: Graham asserts that they do not change the classification except in unusual cases.

Robinson has designed an adapted Bayesian method that is claimed to be a strict improvement on Graham's approach: true Naïve Bayesian is supposed to be better yet, although it often seems to offer only a small improvement in experiments reported here. Graham's success in spam filtering shows that even an extremely simple and less-principled approach to spam feature detec-



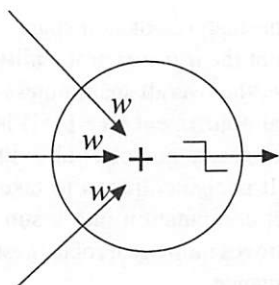


Figure 5: Artificial Neuron

tion and machine learning can get good results in practice.

#### 4.2.4 Perceptron

Neural nets are commonly used in supervised learning. The simplest form of neural net is the single-element *perceptron*: a message is classified as spam if and only if

$$\sum_{i=1}^m w_i \cdot x_i > w_0$$

where the  $w_i$  are real-valued *weights*. Note that the output is made binary by thresholding: in addition to being convenient for a binary classifier, this non-linearity is important in building larger neural networks. Figure 5 shows the perceptron structure schematically.

The weights are assigned during the training phase by gradient descent. Repeated passes are made over all training instances: small adjustments are made to the weights on misclassified training instances until the number of misclassified validation instances is minimized. This somewhat awkward procedure minimizes the probability of *overtraining* the perceptron.

Multilayer neural nets use the output of single perceptrons or similar structures as inputs to subsequent perceptrons. This allows the system to learn more complex features, at the expense of more complex training and difficult control.

Perceptrons and other artificial neurons accommodate binary and discrete features essentially by treating them as continuous. Reinforcement learning in these systems is by adjusting the training weights to correctly reclassify misclassified instances.

#### 4.2.5 ID3

Decision tree learning is a bit more complicated than the above methods. The ID3 decision tree algorithm [17] is a simple, classic decision tree learner. The information-theoretic *entropy*  $U$  of a set of messages  $S$  represents the difficulty of determining whether a message in  $S$  is spam or non-spam:

$$U(S) = -p(S_+, S) \log_2 p(S_+, S) - p(S_-, S) \log_2 p(S_-, S)$$

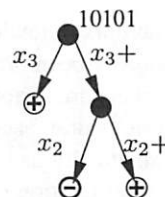


Figure 6: Decision Tree

$$p(S_+, S) \log_2 p(S_+, S) + p(S_-, S) \log_2 p(S_-, S)$$

where

$$p(S_0, S) = \frac{|S_0|}{|S|}$$

If  $S$  is partitioned based on the value of some particular feature  $x_i$ , so that  $S = S_{x_i} \cup S_{\bar{x}_i}$ , the information-theoretic *gain*

$$G(S, x_i) = U(S) - p(S_{x_i}, S)U(S_{x_i}) - p(S_{\bar{x}_i}, S)U(S_{\bar{x}_i})$$

represents the information gained by considering the subsets  $S_{x_i}$  and  $S_{\bar{x}_i}$  separately.

In ID3 the feature  $x_i$  yielding the highest gain on the training set  $T$  is selected for splitting. These subsets are further split until subsets are produced containing instances of only one or largely a single classification. The resulting tree is used in the classification stage: target instances are given the classification matching that of the training instances in their leaf subset. Overtraining is controlled by stopping the split when the largest gain is small, or when the statistical significance of a split as given by a  $\chi^2$  test is too low. Figure 6 shows a binary decision tree: an instance is classified by walking from the root of the tree to the leaf, choosing a direction at each node based on the properties of the given feature. Since the given instance has feature 3 positive and feature 2 negative, it will be classified as ham.

Decision trees can easily accommodate multi-valued discrete features by way of  $n$ -ary trees. Continuous features are usually handled by quantization. Reinforcement learning usually involves simply putting the newly-classified instance at the appropriate leaf: occasionally tree operations may have to be performed to preserve the property of splitting on the highest-gain features first.

#### 4.3 Advanced Learning With Neural Nets

In addition to the relatively unsophisticated techniques described above, more advanced machine learning techniques can also be used to filter spam. In general, these techniques trade off more complex and difficult designs and implementations for potentially higher quality results. This study explores one such approach as an example: constructing a multilayer neural network. This methodology generalizes the simple perceptron of Section 4.2.4, and provides a good illustration of the trade-



offs of an advanced machine learning approach for spam detection.

When constructing a multilayer neural network, one is faced with the choice between implementing from scratch, or attempting to use an existing package that is freely available. Implementing a neural network from scratch is appropriate when performing neural network research: however, it requires substantial effort to develop and debug it, and more effort still to validate it. Subtle numerical bugs can easily contaminate data in ways which are difficult to detect.

Given a focus on spam filtering research rather than neural network research, a free software platform is a natural choice. Using free neural network software leverages years of development and debugging effort. Because free software is in widespread use by researchers around the world, it undergoes intense scrutiny for correctness, and bugs can be fixed quickly when they are found. The availability of the software makes it easier for other researchers to reproduce and extend results. One caveat, however, is that a firm grasp of the principles behind neural networks is still necessary. Neural networks are sometimes finicky learners, and can produce poor results when improperly constructed and used.

### 4.3.1 Introduction to Neural Networks

A neural network consists of multiple, interconnected computational units. Each unit can have multiple inputs, but only a single output. The unit's basic function is to add up the values of its inputs, and transform the result with a nonlinear function to produce its output. The individual units are not very powerful by themselves, but when linked together in a network they can carry out complex computations.

Although each unit can only produce a single output value, this value can be used as input to many other units. Connections between the output of one unit and the input of another are called links. Each link has an associated weight. The output of the first unit is multiplied by this weight to become the input of the second unit. In a network with hundreds of units there can be thousands of links, and therefore thousands of weights. It is these weights which change as the network is trained.

The network configuration used for spam filtering was a basic *feedforward* topology. In this configuration, the network can be viewed as a series of layers of units with the outputs of one layer fully connected to the inputs of the next layer. This topology is called feedforward because there are no loops (links only go forward, never backward) and no jumps (links never skip over intervening layers). The input cascades from layer to layer, undergoing a transformation at each step, until it becomes the output. The first layer of the network is the *input layer*. This layer collects the input and passes it through

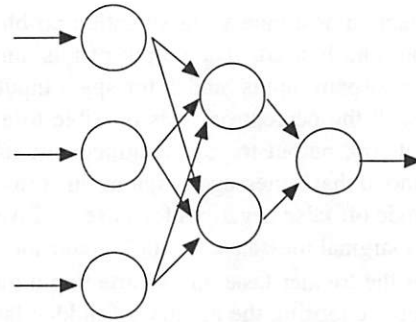


Figure 7: Neural Net

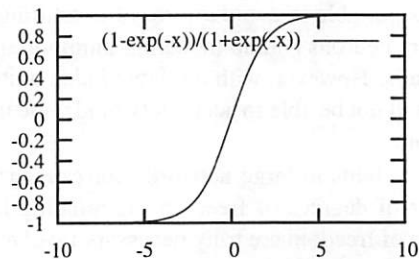


Figure 8: Sigmoid Response Function

weighted links to the first interior *hidden layer*. Each hidden layer computes the weighted sum of its inputs, and then transforms the resulting values with a nonlinear *transfer function* to produce its output. The last hidden layer sends its output to the *output layer*, which in some configurations may apply a final nonlinear transformation. In our network, the output layer did not use a nonlinear transfer function. Figure 7 shows one such network, with three input neurons, two hidden neurons, and one output neuron.

The nonlinearity of the hidden units is what gives a neural network its power to learn complex functions. Without some form of nonlinearity, the transformations performed by the layers of the network amount to nothing more than a series of matrix multiplications; a multilayer network would be equivalent to our simple perceptron. Thus it is necessary to introduce nonlinearity if the network is to be able to learn complex mappings. For our networks we selected a transfer function called the sigmoid, an S-shaped curve whose graph is shown in Figure 8. This function is commonly used in neural network research, and should be available in any free or commercially available neural network package.

### 4.3.2 A Spam Filtering Network

In the feedforward networks employed in this project, the input layer consists of a number of units equal to the feature vector size of the input: each input unit corresponds to a single feature. Because we are treating

spam detection as a binary classification problem, only one output unit is used. The output of this unit should be 0 for nonspam inputs, and 1 for spam inputs. Note that, as with the perceptron, it is possible to allow the neural network output to vary continuously, indicating the likelihood that a message is spam: this provides one way to trade off false negatives for false positives, while flagging marginal messages for further consideration.

One of the trickier tasks in constructing a multilayer neural net is choosing the number of hidden layers and the number of units in each layer. The spam filter design uses a single hidden layer with many fewer units than the input layer. It is desirable to have as few hidden units as possible to avoid a form of overfitting in which the extra neurons end up modeling unimportant training set details. However, with too few hidden units, the network will not be able to accurately model the underlying function.

The weights in large networks correspond to a large number of degrees of freedom. Estimating how many degrees of freedom are truly necessary involves estimating the degree of correlation between inputs. It is easier to estimate the number of degrees of freedom if the inputs are as uncorrelated as possible. A word clustering feature detector (Sec. 5.3) helps to achieve this by dividing words into clusters which are maximally independent.

Like the perceptron, a neural net is trained through a process of adjusting the link weights between layers so as to bring the actual output vectors closer to the desired ones. The network essentially learns "by example." The most famous training algorithm for feedforward networks is backpropagation. In backpropagation, values first flow forward through the network to produce outputs. These outputs are compared with the desired ones, and errors are then propagated backward through the network, adjusting the weights so as to reduce the error. A closely related training technique, Rprop, has many of the same features as backpropagation: it was selected for filtering because of its robustness and quick convergence.

Several issues arise when selecting a network design. It is clear that the input layer must have as many units as the number of features in the feature vectors, but these values can be presented to the network in different ways. Simplest is to treat each feature as a binary input, which is 0 when the feature is not present, and 1 if it is, regardless of how many times that feature might occur in a single message. This makes the neural net compatible with the other machine learners discussed here, and simplifies feature processing.

(Another option would be to standardize the input values according to some statistical model. Originally, the input values were standardized under the assumption

that they are normally distributed. The purpose of this was primarily to accelerate the learning of the network. Because the feature counts have a wide spread—many are 0, some can be in the hundreds—it takes a long time for the network to adjust its weights to account for this spread. Standardizing the inputs brings the values closer together, which speeds convergence.)

### 4.3.3 SNNS

SNNS is a neural network simulation package developed at the University of Stuttgart in Germany. The source code is open, and the software is freely available for research and academic use. The homepage of the SNNS project can be found at <http://www-ra.informatik.uni-tuebingen.de/SNNS/>.

SNNS was selected for many reasons. Most important is its great flexibility. SNNS supports a wide range of network topologies, not just feedforward networks. It provides an array of training algorithms which can be applied to almost any kind of network design. Parameters like the number of layers, layer dimensions, links, and transfer functions are all fully configurable. This allows experimentation with a variety of designs with no time wasted recoding the network.

SNNS has a very attractive GUI which runs under X. Although there is no visual design tool, it is straightforward to configure a basic feedforward network. The GUI can display the network in action and can produce graphs of output error over time. These features make it simple to visually determine when the network is performing well.

The package can also be operated in a batch mode. The batch interpreter has a complete scripting language to automate training sessions. Training and validation can be scheduled arbitrarily, and error results can be written to disk at any time during the process. This makes it simple to begin training runs on large datasets overnight.

Finally, SNNS has the ability to translate a trained network into a C program. Although the network cannot be trained further once converted to C, it is very compact and easily callable from other C code. This makes it possible to build high-performance message classifiers once the network is trained.

The use of an open source neural network "construction kit" thus permits simple implementation of a quite sophisticated machine learner for spam. A similar approach could be employed for other machine learners discussed earlier, as open source construction kits are available for a wide variety of machine learning techniques.

## 5 Feature Detection

The problem of feature detection is largely orthogonal to the problem of learning on the identified feature set. (This observation does not seem to be commonly made in the open source community, and deserves wider attention.) This work has experimented with several different types of feature detector. More sophisticated methods have been applied to feature detection. For example, Lewis' Data Enrichment Method [13] is unbelievably powerful, but has other drawbacks that prevent its use in the real world.

### 5.1 SpamAssassin

The feature set computed by SpamAssassin was the initial basis of this study. The advantages of this approach are manifold: SpamAssassin provides several hundred hand-crafted binary features, the features seem to be reasonably sensitive, and using SpamAssassin features permits easy comparison with the classification performed by SpamAssassin itself.

The features recognized by SpamAssassin provide a fine feature source for the learning algorithms described above. Maintenance of this feature detector, however, is a tremendous amount of work. In addition, the detector is quite slow, as slow as a few messages per second if the network-based lookup features are enabled.

### 5.2 Gain-Based

Rather than hand-crafting a feature detector, it would be useful to automatically extract features directly from the corpus. When trying to classify email the most natural features are the words of the message. A standard approach is to use individual words directly as features. Each feature vector element indicates whether a particular word is present in a message. However, the English language contains thousands of words, while every email message contains only a small subset of those words. This makes it difficult to decide which words are good representatives of spam mail and nonspam mail, given a limited amount of features.

To this end, a feature detector has been constructed that selects e-mail body words with the highest information-theoretic gain (Sec. 4.2.5) as likely high-utility candidates for learning algorithms. This detector appears to work quite well, with learning accuracies approaching those achieved with the SpamAssassin detector. The detector operates by breaking the e-mail body into words using simplistic rules, and then measuring the gain of each word using a dictionary. Those words with gain above a set threshold are retained.

### 5.3 Word Clustering

A preliminary attempt was made to assess the performance of word-based features, using the 134 words with

the highest information gain as features. The resulting feature vectors were quite sparse: most messages had few high-gain words. The training cost of supervised learning algorithms generally grows in proportion to the number of inputs. For example, the number of weights in a feedforward neural network grows in this fashion. Because training time is proportional to the number of weights, considering a large number of inputs can make training intractably slow. What is needed, then is a way to automatically extract a small number of features that nonetheless give a strong signal on every message.

One solution to this problem is to cluster words of similar meaning together into a single feature. This allows more unique words (thousands instead of hundreds) to be considered when scanning a message for features, while simultaneously keeping the feature count low enough to make training manageable. This gives much better coverage of the set of words occurring in email messages.

An information-theoretic clustering algorithm described in a recent paper by Dhillon and Modha [4] seems to be a good candidate. This algorithm was selected from among many other clustering algorithms proposed for machine learning primarily because its execution time is linear in the number of words and clusters. Other clustering techniques tend to be quadratic or worse in the number of words and clusters. In addition, the Dhillon algorithm is unlike some other clustering algorithms in that it considers the spam/non-spam classification of the messages while clustering. The mechanics of the Dhillon algorithm are briefly described here; the original paper by Dhillon gives a deeper look at the information-theoretic concepts underlying it.

The initial assignment of words to clusters is done by allocating two sets of empty clusters of equal size. Words which occur more often in nonspam messages are randomly assigned to one of the "ham clusters". Similarly, spam words are assigned to a spam cluster. This concept of nonspam clusters vs. spam clusters is only meaningful during initialization, since the clustering algorithm will move the words between clusters to minimize the clustering metric. In the final clustering, some clusters may contain both ham and spam words.

After the initialization step the algorithm becomes iterative. On each pass through the loop words are moved between clusters to decrease the value of a divergence metric. This metric quantifies the average dissimilarity of the words in each cluster. Minimizing this value thus maximizes the intra-cluster similarities. Iteration then continues until the divergence metric does not change more than a fractional amount from the previous iteration.

The implementation of the clustering algorithm is done in three major blocks: collection of data, cluster-



ing of data, and output of clusters. Each message body is initially scanned and tokenized into words. The clustering function takes as input the desired number of clusters. The clusters are initialized as described above. To reduce the number of candidate words to a manageable size, the implementation only considers words which occur more than a minimum number of times over all the messages.

The iterative algorithm then executes. After a number of iterations the clustering converges. The number of clusters output by the algorithm may be less than requested, because some clusters become empty during the iteration of the main loop. Only non-empty clusters are output at this phase.

The final step is generating the feature vectors. Using the cluster file generated by the clustering algorithm, each message body in the corpus is again scanned, matching each scanned word to its cluster. This is done by loading the cluster file into a hash table which maps each word to its cluster number, which is a nearly constant-time operation. Since each word in the message must be examined once, the time it takes to scan a message is thus roughly linear in the number of words in the message.

## 5.4 Combined Approaches

It is sensible to consider combinations of feature detectors. The comparison is complicated. As noted previously, larger feature sets slow recognition and learning; it may be better to use more features of a given type than to combine features of several types.

For the feature detectors examined here, combinations are less problematic. The SpamAssassin detector operates on header information (and by all accounts does well at this): the other detectors operate only on body information. The clustering detector is believed to perform strictly better than the gain-based detector, since it is essentially a superset of it. Thus, there are five combinations that are leading candidates for examination: the three detectors alone; SpamAssassin plus gain-based; and SpamAssassin plus clustering.

## 6 Related Work

Work on text classification in general, and spam detection in particular, dates back many years in the machine learning community. For example, Androutsopoulos has worked with a number of researchers on machine learning spam filters [1, 19]. A good overview of machine learning for e-mail classification is in Itskevitch's M.S. Thesis [9].

These approaches first caught the wide attention of the open source community with Graham's web article (Sec. 4.2.3). This article and Robinson's commentary on it inspired a number of implementations of semi-

Bayesian word-based filters, many of which can be found at [sourceforge.net](http://sourceforge.net).

At the same time, non-learning-based approaches to spam filtering have also been widely attempted. The SpamAssassin tool is a freely-available Perl-based spam filter that combines hand-crafted features using a perceptron. Initially, the perceptron weights were hand-tuned: more recently, a genetic algorithm was used to train the weights on a synthetically composited corpus. Oddly, the SpamAssassin authors have apparently not used a traditional gradient-descent approach to tune their perceptron: it was this omission that inspired the authors of this paper to begin the research reported here. SpamAssassin combines its primary feature data with other sources of information, such as spam databases and word data, to produce a final classification.

## 7 Corpora

The problem of selecting a corpus for evaluation of learning algorithms for spam detection is a difficult one. One challenge is that private e-mail is rarely available for public study: thus, other sources of ham must be found if the corpus is to be made publically available. For evaluation purposes, four corpora were assembled.

The first corpus consists of the first author's e-mail over a recent two-year period, a total of 15,498 messages. These messages were randomly sampled to select exactly 15,000 messages for ease of use. This corpus has the advantage of verisimilitude: most studies have used only corpora consisting of synthetic combinations of messages from public mailing lists and spam databases. The corpus is about 50% spam: a percentage higher than indicated in older publications on the subject [3], but consistent with current anecdotal evidence. For privacy reasons, the corpus itself is not publically available: however, the instance data derived from the corpus by feature recognition is.

The second corpus is a total of 15,000 messages drawn equally from two sources: 50% of the messages are ham from the X Window System developer's Xpert mailing list; 50% are spam from the Annexia spam archive [2]. The *a priori* accuracy rate of this corpus is much lower; there are frequent classification errors in both data sets. This corpus is publically available.

The third corpus is the Lingspam corpus, a synthetic corpus of 2405 messages. 80% of these messages are ham from a linguistics mailing list: the rest is spam. The corpus has been made publically available by Ion Androutsopoulos, and used in publications by several authors: it thus provides a basis for comparison with published work.

The fourth corpus is used to tune SpamAssassin, and consists of 8686 messages, 80% ham and the remainder spam, from a variety of sources. It is deemed useful for



Table 1: Percentage Error Rate In Classification

	SA		BD		SA+BD		CL		SA+CL	
	⊕	T	⊕	T	⊕	T	⊕	T	⊕	T
synthetic (7500 ham, 7500 spam)										
hamming	0.40	7.11	2.10	4.45	1.00	4.86	1.29	2.56	0.68	2.80
nbayes	0.10	8.12	0.12	11.30	0.02	8.86	0.52	3.98	0.14	3.08
graham	1.70	5.98	0.52	9.70	0.76	3.74	0.00	47.64	0.06	9.50
neuron	1.60	5.90	2.66	4.40	1.90	3.30	1.40	2.58	1.84	2.68
dtree	0.53	6.02	3.33	5.41	2.08	4.23	2.20	3.86	1.88	3.52
net	0.30	5.28	1.04	6.38	0.48	3.16	0.98	2.68	0.54	3.28
personal (8400 ham, 6600 spam)										
hamming	0.36	3.27	1.33	5.44	0.59	2.66	1.14	2.49	0.29	2.89
nbayes	0.46	2.46	2.56	14.48	1.26	4.56	2.96	5.42	0.60	1.94
graham	3.58	4.36	3.94	9.06	4.56	5.44	0.00	38.14	0.14	3.32
neuron	1.32	2.28	2.04	6.84	0.76	2.08	1.24	2.72	0.96	1.98
dtree	0.58	2.24	2.23	6.53	1.14	2.51	1.91	3.68	1.46	2.76
net	0.28	2.20	0.90	6.10	0.48	2.08	0.86	2.68	0.40	2.06
lingspam (1924 ham, 481 spam)										
hamming	0.87	9.86	0.87	5.24	0.78	7.05	0.78	4.06	0.44	4.87
nbayes	0.25	5.99	0.87	5.24	0.50	4.49	3.25	5.12	1.50	3.75
graham	0.87	7.74	1.25	2.62	0.75	3.25	3.00	6.24	2.50	5.12
neuron	1.87	5.49	1.87	3.75	0.75	3.25	9.11	11.61	8.49	10.74
dtree	0.94	8.24	2.12	6.12	1.03	5.06	1.37	4.24	1.37	3.50
net	1.12	6.10	0.87	3.86	1.00	4.36	0.50	3.11	0.25	2.24
spamassassin (6948 ham, 1737 spam)										
hamming	0.16	3.28	1.00	6.22	0.15	2.95	1.06	3.30	0.30	3.13
nbayes	3.90	5.35	4.73	13.75	4.66	7.22	6.80	10.02	5.08	6.42
graham	5.01	6.15	10.50	13.26	6.98	7.56	7.50	9.64	6.04	6.77
neuron	1.00	2.14	2.49	9.95	0.76	2.38	7.25	10.12	9.46	11.88
dtree	0.59	2.66	1.68	6.48	1.73	2.80	1.75	3.30	0.86	2.25
net	0.62	2.59	1.07	7.15	0.28	2.59	0.62	3.49	0.38	2.38

comparison with a fielded freely-available spam filter, as well as being a robust corpus useful in its own right.

## 8 Evaluation

Each of the algorithms reported in Sec. 4.2 has been implemented in C and Perl by the authors. These implementations are freely available as noted at the end of this document. Accuracy and speed of the implementations have been measured on a 1.8AGHz AMD box with 512MB of main memory, running Debian "Woody" with kernel 2.4.19.

The preliminary nature of the measurements reported here should be emphasized. There are an enormous number of interesting experiments that can be run given the sample setup, and there is an enormous amount of work that can be done to improve the design and implementation of these spam filters. Nonetheless, the measurements in this section serve both to provide a gross comparison between various learners and detectors, and to illustrate some of the issues that arise in practical ma-

chine learning. Figures 1, 2 and 3 show some of the measurements made. Those measurements are good illustrations of the power of experimental data in elucidating machine learning issues.

Table 1 shows the classification accuracy of the algorithms on the sample corpora. The columns labeled ⊕ and T are false positive and total error percentages respectively. (The false positives are shown as percentage of total messages, rather than percentage of ham messages; they thus reflect the mix of messages in the given corpus.) The columns labeled SA, BD, and CL denote the use of the SpamAssassin, Body Dictionary, and CLustering feature detector: the SA+BD and SA+CL columns denote the use of combined feature sets. All programs were run using default parameter settings. Statistics reported are worst of 10 runs, in accordance with PAC theory [21]. Two-thirds of the corpus have been used for training (and validation when required), while the final third has been used for classification: a new random split has been used for each run, with all

programs being run over the same 10 splits. The gain threshold of the body word detector was selected to give a maximum of a few hundred features across all corpora. This seemed to give sufficient accuracy for evaluation purposes, but more experimentation in this area would be prudent.

Table 1 is unfortunately difficult to read. Nonetheless, it contains a great deal of useful data, from which several conclusions can be drawn. While the differences between classifiers and between feature detectors are quite significant, it can fairly be said that overall the accuracy of the filtering systems is similar. The exceptions reveal a number of interesting phenomena.

The more complex classifiers seem to be consistently better than the simpler ones. In particular, the neural net is the most consistently strong classifier: the decision tree learner also produces good results. As expected, the combined feature detectors tend to be stronger overall than their components: the SA detector appears to work well with most classifiers and corpora.

Specific classifiers seem to have trouble with specific corpora or detectors. Note particularly the 100% false negative rate of the graham classifier on the personal and synthetic corpora with the CL detector. It is believed that this anomaly is not a program defect, but results from a peculiarity of the CL feature set: for these large inputs, the detector tends to group all of the spam words into just one cluster, while the ham words have a large number of clusters. Graham's heuristic does not cope well with this case: the large number of ham features swamps the signal from the much more significant spam features. The graham classifier appears to be less strong than the nbayes classifier overall, but not dramatically so: the choice of corpus and features appears to matter significantly. The hamming detector appears to be a good, reliable detector overall, and may actually be a reasonable choice in situations where its slow classification rate can be reduced or ignored.

Table 2: Feature Detection Time

	s/Kmsg	s/MB
SA	1784	391
BD	15.2	1.9
CL	14.0	1.7

Table 2 shows feature detection time for the synthetic corpus. The BD classifier gain threshold is 0.05. Times shown are wall clock seconds per 1000 messages and seconds per megabyte (1,048,576 bytes). Neither the BD nor the CL detector is significantly optimized for performance—significant improvements could be expected in practice. These times suggest why work on alternate feature detectors and an over-

all move away from SpamAssassin may be important in the future.

Table 3: Training and Classification Time

	T	C
hamming	0.00	44.05
nbayes	0.65	0.06
graham	0.65	0.02
neuron	10.66	0.00
dtree	5.45	0.00
net	140.35	0.13

Table 3 shows training and classification time for the synthetic corpus and SA feature detector. Training and classification times are exclusive of feature detection and other times. All times are CPU seconds per 1000 instances, and are the average of 10 runs. Times shown as 0.00 are less than 0.01 seconds per 1000 instances, in other words in excess of 100,000 instances per second. As expected, the neuron and dtree detectors require a moderate training period. The net detector is slow to train (although not unusably so). The hamming detector as implemented is probably too slow to use for server filtering, although it would work fine for filtering an individual's messages.

## 9 Deployment

The initial integration target for the work described here has been SpamAssassin, a rule-based mail filter written in Perl by a team including Justin Mason. SpamAssassin is an open source project distributed under Perl's Artistic license: it was hoped that it would be a good basis for third-party extension. The mechanism by which SpamAssassin classifies a message is via handcrafted header and body text analysis rules, along with blacklist/whitelist support (lists of addresses to automatically deny or accept) and use of a spam tracking database such as Vipul's Razor. After the test suite has been run, the mail message can optionally be marked with a spam header for easy processing by a user's mail reader. The package is primarily made up of libraries of test and mail handling code, forming an API that allows for easy integration with mail applications on multiple platforms. Various command line and daemon scripts for interfacing with the API are also supplied.

The SpamAssassin API libraries consist of two main sections: code to run each of the tests, and the engine that calls the test code and combines (and scores) the results. The text analysis portion of the testing code is comprised of regular expressions that are matched against the headers and body of the mail message. SpamAssassin allows for quite a bit of flexibility by coding many of the tests in user editable configura-

tion files—tests can thus be added or modified without change to the Perl libraries themselves. Unfortunately, the format of the configuration file only allows for the specification of new tests involving single regular expressions.

The classification engine is responsible for handling mail input and output, including mail message headers and body text. It also supports routines to update test parameters. For example, adding and removing mail addresses from accept and deny lists, or reporting a mail message to collaborative spam tracking databases online. After running the full suite of classification tests, the testing engine scores the mail message based on the results of each of the tests. If the message is classified as spam the engine takes appropriate action, such as rewriting the mail message to include easily recognizable tags for the user in the subject line and message body, as well as adding an extra header for the user's mail user-agent to automatically refile spam messages.

Recently, the SpamAssassin team has added support for classification of mail messages using a “Bayesian-like form of probability-analysis”, apparently based on a Graham/Robinson (Sec. 4.2.3) or Naïve Bayes detector. This extension seems to allow for online learning and storage of new message characteristics, although the documentation is incomplete at the time of this writing. While the Bayesian extension appears to implement the same algorithm as the Naïve Bayesian classifier described here, it is embedded in the SpamAssassin code. This makes it hard to inspect, and requires modifying the SpamAssassin code itself to make changes.

One straightforward way to integrate the classifiers with the SpamAssassin package is to convert them to Perl. Currently the perceptron learner has been completely converted and an offline classification mode has been implemented. The SpamAssassin feature set is used as training input to this learner. The other learners are only partially integrated: the classification phase is available to SpamAssassin. A SpamAssassin-style manually-weighted perceptron is used to integrate the classifiers, including SpamAssassin's built-in classifier. New feature detectors can be integrated with the setup and used by all of the learning classifiers.

Modifications to the SpamAssassin code have been kept as minimal as possible. Code implementing a linear perceptron has been added to the `check()` method of the `Mail::SpamAssassin` module to allow as many classifiers to be run on a message as desired: the result of each classifier is weighted in a user-specified fashion. Parameters have been added to the `new()` method of the module to select combinations of classifiers and feature detectors, weights for the classifier's output, and a perceptron threshold value used to make the final classification decision on the e-mail message.

Modifications have also been made to the `Mail::SpamAssassin::PerMsgStatus` module. Parameters have been added to the `new()` method specifying the existing (and allowable) feature detectors and classifiers. Each feature detector is associated with the method implementing it as well as the dictionary file required to create a feature vector. Each classifier is associated with a file containing state information required for classification (a listing of current weights for the single neuron learner, for example). Methods have been added to implement the feature detectors, or retrieve the classification test results from SpamAssassin's builtin feature detector, as well as to encode the returned feature sets according to the specified dictionary. A `_psam_check()` method has also been added: the method calls the indicated classifier with the created feature vector and returns the result.

As of this writing, the integrated system is close to being ready to submit to the SpamAssassin team for review and integration. Unfortunately, the runtime overhead associated with the Perl implementation of SpamAssassin and the learners and classifiers has proven to be a significant problem. Thus, the direction to take from here is unclear. Finding a simpler and more efficient open source framework is currently under consideration as an alternative, as is building yet another mail classification framework.

## 10 Future Work and Conclusions

It has been said that good research raises more questions than it answers. By that standard, the research reported here has been successful indeed. Much more work is needed on the corpora: it would be nice to establish a trustworthy and representative test corpus of about 10,000 messages for future work. Validation and tuning of both the feature detectors and classifiers is badly needed to establish confidence in their correctness and to understand ideal operating parameters for them. Using ensembles of detectors, detector biasing techniques, and other advanced methods should be explored to improve accuracy. An integrated mail-filtering system should be built, and overall system accuracy and performance evaluated. An anonymous referee of this paper suggested that the learners and classifiers should be packaged in a library for use in this and other projects: this is an excellent idea and will be implemented.

Supervised machine learning is an effective technique for spam filtering. The methods described in this paper provide the basis for reasonably accurate, efficient classification of messages as ham and spam. Freely-available software implementors interested in spam filtering are encouraged to take advantage of these techniques (and their more sophisticated cousins) to help control the spam deluge.



## Acknowledgements

Special thanks to: the Usenix Association and the Portland State University Computer Science Department for enabling the student authors of this paper to attend the conference; Carl Worth for shepherding the paper; Keith Packard and Mike Haertel for help and advice (as usual) with code and algorithms; and Jeff Brandt for valuable ideas, criticism, encouragement, and help with corpora.

## Availability

The instance data and implementations used in this work are freely available under an MIT-style license at <http://oss.cs.pdx.edu/psam>.

## References

- [1] Ion Androutsopoulos, John Koutsias, Konstantinos V. Chandrinou, George Paliouras, and Constantine D. Spyropoulos. An evaluation of naive Bayesian anti-spam filtering. In G. Potamias, V. Moustakis, and M. van Someren, editors, *Proceedings of the Workshop on Machine Learning in the New Information Age: 11th European Conference on Machine Learning*, pages 9–17, Barcelona, Spain, June 2000.
- [2] Annexia Spam Archive. URL <http://www.annexia.org/spam/index.msp> accessed 25 November 2002, 22:00 UTC.
- [3] Lorrie Faith Cranor and Brian A. LaMacchia. Spam! *Communications of the ACM*, 41(8):74–83, August 98.
- [4] Inderjit Dhillon, Subramanyam Mallela, and Rahul Kumar. Enhanced word clustering for hierarchical text classification. Technical Report TR-02-17, Department of Computer Sciences, University of Texas at Austin, Austin, Texas, March 2002. URL <http://citeseer.nj.nec.com/507920.html> accessed 8 Apr 2003, 00:41:35 UTC.
- [5] Matthew L. Ginsberg. *Essentials of Artificial Intelligence*. Morgan Kaufmann, 1993.
- [6] Paul Graham. A plan for spam. URL <http://www.paulgraham.com/spam.html> accessed 18 November 2002, 22:00 UTC.
- [7] Paul Graham. Better Bayesian filtering. In Houbart [8]. URL <http://spamconference.org/proceedings2003.html> accessed 23 March 2003, 21:18 UTC.
- [8] Gilberte Houbart, editor. *Proc. 2003 MIT Spam Conference*, Cambridge, MA, January 2003. URL <http://spamconference.org/proceedings2003.html> accessed 23 March 2003, 21:18 UTC.
- [9] Julia Itskevitch. Automatic hierarchical e-mail classification using association rules. Master's thesis, Simon Fraser University, July 2001.
- [10] Paul Judge. Spam research: Establishing a foundation and moving forward. In Houbart [8]. URL <http://spamconference.org/proceedings2003.html> accessed 23 March 2003, 21:18 UTC.
- [11] Seth Kaplan. How antispyware software works. *Wired Magazine*, 11(4):43, April 2003.
- [12] David D. Lewis. (Spam vs. forty years of machine learning for text classification. In Houbart [8]. URL <http://spamconference.org/proceedings2003.html> accessed 23 March 2003, 21:18 UTC.
- [13] Henry R. Lewis. The data enrichment method. *Operations Research*, vol. 5, 1957. Reprinted in *J. Irreproducible Results* 15(1), 1966, and in subsequent collections from same.
- [14] Stanton McCandlish. Archeology of spam. URL [http://www.eff.org/Spam\\_cybersquatting\\_abuse/Spam/archeology\\_of\\_spam.article](http://www.eff.org/Spam_cybersquatting_abuse/Spam/archeology_of_spam.article) accessed 8 April 2003, 20:45 UTC.
- [15] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [16] Marco Paganini. ASK: Active Spam Killer. In *Proc. 2003 Usenix Annual Technical Conference*, San Antonio, TX, June 2003. To appear.
- [17] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(81):106, 1986.
- [18] Gary Robinson. Spam detection. URL <http://radio.weblogs.com/0101454/stories/2002/09/16/spamDetection.html> accessed 18 November 2002, 22:00 UTC.
- [19] G. Sakkis, I. Androutsopoulos, G. Paliouras, V. Karkaletsis, C. D. Spyropoulos, and P. Stamatoopoulos. Stacking classifiers for anti-spam filtering of e-mail. In L. Lee and D. Harman, editors, *Empirical Methods in Natural Language Processing (EMNLP 2001)*, pages 44–50, Pittsburgh, PA, 2001.
- [20] SpamAssassin. URL <http://www.spamassassin.org> accessed 18 November 2002, 22:00 UTC.
- [21] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.



# Network Programming for the Rest of Us

Glyph Lefkowitz  
*Twisted Matrix Labs*

glyph@twistedmatrix.com, <http://www.twistedmatrix.com/users/glyph>

Itamar Shtull-Trauring  
*Zoteca*

itamar@zoteca.com, <http://www.itamarst.org/>

## Abstract

Twisted is a high-level networking framework that is built around event-driven asynchronous I/O. It supports TCP, SSL, UDP and other network transports. Twisted supports a wide variety of network protocols (including IMAP, SSH, HTTP, DNS). It is designed in a way that makes it easy to use with other event-driven toolkits such as GTK+, Qt, Tk, wxPython and Win32. Implemented mostly in Python, Twisted makes it possible to create network applications without having to worry about low-level platform specific details. However, unlike many other network toolkits, Twisted still allows developers to access platform specific features if necessary. Twisted has been used to develop a wide variety of applications, including messaging clients, distributed hash tables, web applications and both open source projects and commercial applications.

## 1 Introduction

Networked infrastructure development currently exhibits a curious asymmetry. Very high-level infrastructure, such as web servers, and very low-level infrastructure, such as OS-level I/O, have both been actively developed. However, there is no popular middle layer between the two; each high-level abstraction implements all the infrastructure from the base OS level all the way up to its application needs in a specific way.

Development in the area of high performance multiplexing has continued on many platforms, yielding ever-increasing performance. These mechanisms

are many and varied: `/dev/poll`, `epoll`, `select`, `poll`, `kqueue[1]`, completion ports, and POSIX AIO, to name a few. On the other end of the spectrum, many frameworks provide high-level constructs for specific application domains. For example, web application servers provide infrastructure for developing HTTP-based applications. There are few frameworks that provide access to both the low-level and high-level infrastructure required in real networking applications. Web application servers don't provide access to their low-level networking event loop for extension with new protocols, while low-level libraries require far too much work to be usable out of the box.

Twisted is a networking framework suitable for building a wide range of networked servers and clients. Twisted provides portability by using high-level abstractions of protocols, various transports (such as TCP and UDP) and an event loop, allowing deployment of the same code across multiple platforms, primarily Unix and Windows NT.

However, access to platform-specific functionality is a real requirement for many applications. For example, without the ability to access file descriptors directly, it isn't possible to write programs that integrate access to the serial port into the event loop.

Twisted provides low-level access to operating system and event loop specific functionality. On UNIX, for example, it is possible to register file descriptors with `select` and `poll`, even though this functionality is not available on Windows. At the same time, one can use the Win32 API to access a serial port through different mechanisms when using Windows. Twisted makes it possible for a user to abstract such a feature themselves if the framework does not pro-

vide it already.

Twisted also provides many high-level facilities commonly used by networking applications. A mail server shares a large number of requirements with a web server, such as I/O, protocol parsing, logging and daemonization. In addition, many standardized protocols are shared between applications, e.g.: web mail requires both HTTP and SMTP. Including basic implementations of such protocols saves the developers the need of developing them from scratch.

By providing the full spectrum of functionality for networking applications, both low-level and high-level, developing networking applications is a far simpler task, allowing the developer to concentrate on developing their application rather than reinventing the wheel. Twisted also provides a middle layer so that high-level networking applications may take advantage of low-level advances in functionality and scalability.

This approach contrasts to the two main approaches taken by most networking frameworks. One approach is to use a low-level framework and language (C or C++). The low-level approach gives access to all of the capabilities and APIs provided by the operating system, and if done correctly can result in a very fast program. On the other hand, pervasively using platform-specific functionality results in a platform-specific program, so portability is hindered. Unless carefully audited, C and C++ code is more prone to buffer overflows and system-crashing bugs than their high-level counterparts. This results either in a much longer development process as testing locates all the problems, or in a more fragile system.

Another approach is to provide functionality which only provides access to the lowest common denominator between all supported platforms. This approach is taken by most high-level frameworks. While the "lowest common denominator" approach makes the framework very portable, it means one can't do many basic tasks that only work on specific platforms. For example, the Java platform, which takes this approach, does not support running a server as a daemon on UNIX or an NT Service on Windows, since neither of these features are available on other platforms. This decision pleases no-one by trying to please everyone.

High performance is not a major goal of the Twisted

framework. While efficiency and scalability are taken into account during development, flexibility and clean design are more important. This focus stems from the belief that in the real world, as long as performance meets the user's requirements, other factors are more important when choosing a platform. For example, there are a large number of open source and commercial web servers and academic papers describing architectures all of which are significantly faster than the Apache web server. Nevertheless, as of February 2003 Apache runs more than 60% of the sites on the web[3].

To achieve scalable performance that will meet most user's requirements, Twisted uses multiplexing for all I/O operations, and a single thread for almost all computation. As the progression of the Java language has shown[2], blocking, threaded I/O libraries simply do not scale to meet more than the most basic demand. In addition, performance costs associated with context-switching and synchronization, which are exacerbated by Python's global interpreter lock, are eliminated. As others have shown[11], threading is useful only in certain circumstances and should be regarded as a low-level tool. As with all such tools, Twisted has a high-level wrapper that provides a portable and convenient way to integrate threaded code with the event loop when necessary.

Twisted is implemented mostly in Python, with a few parts also available as C extension modules for performance reasons. Python is a very high-level programming language with a great deal of run-time flexibility that allows rapid development of dynamic systems. Despite its high-level nature, it offers access to many system calls necessary for networking, such as `select()`, `socket()` and `poll()`. It provides an excellent base for porting Twisted's functionality to new operating systems.

Thanks to the facilities provided by Python, Twisted is rather small: at the time of this writing Twisted has only 89,000 lines of code in Python, including all of its protocol implementations. The C portion, which only duplicates certain performance-critical python functionality, is even smaller, only 4,000 lines of code in C.

Twisted is currently at version 1.0.3, and is being used by a variety of commercial and open source applications. It is a mature project with a large and active community.

## 2 The Python Programming Language

Python is a high-level object-oriented programming language, with runtimes written in C and Java (and an experimental .NET CLR runtime). Python is highly portable, and runs on a large number of operating systems — including UNIX and UNIX-like systems, Windows and others. However, unlike Java, Python does not go down the path of the the lowest common denominator. Instead, Python supports platform specific features in addition to common functionality. On Windows, Python can be used with COM and Win32 APIs. On UNIX, Python has access to a large range of the POSIX functionality, from `fork()` to signals. At the same time, where necessary Python provides common wrappers for low-level functionality — threads in Python use a common API that provides the same functionality on different platforms.

Because of its high-level orientation, Python alleviates the need to deal with memory allocation, array bounds checking and pointers, speeding development and preventing common security issues. Moreover, Python scales upwards when designing complex systems, allowing well designed libraries to provide powerful functionality with simple interfaces.

Twisted builds on Python's flexibility, power and clean design. Python wraps GTK+, Qt, Tk, wxPython and Win32 GUI toolkits, and thus allows Twisted to integrate with these toolkits' event loops. Python's support for sockets, `select()` and other low-level APIs are wrapped to create Twisted's networking core. Additional Python libraries which wrap C code are also used for various functionality (PyOpenSSL for SSL and TLS, PyCrypto for cryptographic algorithms and so on). Additional low-level functionality is easy to integrate due to the simplicity of extending Python with C or C++.

## 3 The Event Loop

The event loop is the core of Twisted. It implements a pluggable interface to OS-level functionality such as networking, timers and certain commonly available utilities such as SSL encryption. There are two ways of implementing networking

event loops. In one approach, event handlers are called in response to readable or writable events on sockets and then an attempt is made to read or write as much as possible. This method is commonly used with non-blocking sockets. The other approach used is fully asynchronous I/O, where event notification happens when a read or write is finished (e.g. POSIX AIO or Windows' I/O Completion Ports). Currently Twisted implements several non-blocking event loops. The event loop APIs are designed to accommodate fully asynchronous I/O as well, but as yet, no implementations have been released.

An object that implements an event loop in Twisted is called a "reactor". Twisted provides reactors that run on top of `select()`, `poll()`, `kqueue` and Win32 Events. Additionally, it provides reactors that use these same low-level mechanisms, but access them through the APIs of the graphical toolkits, such as GTK+ and Qt. This enables Twisted to run within a graphical application written with these toolkits with no performance impact.

For toolkits which do not provide networking APIs, such as Tk and wxWindows, Twisted provides support modules which will run any reactor at brief intervals as the GUI event loop is running.

On Jython (an implementation of Python written in Java) Twisted provides a Java API based reactor that emulates an event loop using threads.

The reactor implementation may be chosen at runtime, depending on which ones are available and what functionality is required. A Twisted reactor object implements functionality for working with at least some of the following systems:

- TCP
- SSL/TLS
- UDP
- multicast
- Unix sockets
- generic file descriptors
- Win32 events
- process running
- scheduling
- threading.

For example, all of these interfaces are supported on Unix when using the default `select()` based reactor, except Win32 event support. However, when running on Windows, the same reactor will not support generic file descriptors and Unix sockets.

This support for reactor-specific functionality does not mean that all applications written with Twisted are not portable. The programmer can choose to use platform specific functionality (e.g. use the file descriptor support on Unix to write a curses-based console interface), or to use only those interfaces that are cross-platform and supported in all reactors (e.g. use TCP support to write a telnet-based console interface). The choice is made by the programmer, not the framework.

## 4 Networking

Twisted has a small networking core, `twisted.internet` (a Python package), which aims to provide a highly portable socket multiplexing API. This API is based around four fundamental principles.

1. All methods should be named in platform neutral, self consistent and semantically clear ways.
2. The API should be as small and abstract as possible.
3. Low-level functionality should not be disabled or obscured in any way.
4. The same events should be provided from multiple different sources, to allow the same objects to communicate with any semantically identical objects.

Each of these four features has an important impact on the resulting framework's ease of use and implementation. Each will be explored in detail.

### 4.1 Method Naming

While this may sound like a small detail, clear method naming is important to provide an API that developers familiar with event-based programming can pick up quickly. Obscure names like "kqueue"

and `"/dev/poll"` litter the multiplexing landscape, which can make the already-intimidating concept of event-based programming seem even more arcane.

Since the idea of a method call maps very neatly onto that of a received event, all event handlers are simply methods named after past-tense verbs. All class names are descriptive nouns, designed to mirror the is-a relationship of the abstractions they implement. All requests for notification or transmission are present-tense imperative verbs (see Fig. 1).

The naming is *platform neutral*. There is no reference to `select`, `poll`, `kqueue`, completion ports, or even multiplexing. This means that the names are equally appropriate in a wide variety of environments, as long as they can publish the required events.

It is *self-consistent*. Things that deal with TCP use the acronym TCP, and it is always capitalized. Dropping, losing, terminating, and closing the connection are all referred to as "losing" the connection. This symmetrical naming allows developers to easily locate other API calls if they have learned a few related to what they want to do.

It is *semantically clear*. The semantics of `dataReceived` are simple: there are some bytes available for processing. The semantics remain the same even if the lower-level machinery to get the data is highly complex.

### 4.2 Small, Abstract API

The methods described above are all extremely abstract, high-level versions of their OS-centric cousins. This allows Twisted to support a wide range of platforms, even those that differ on such fundamental details as statefulness of the low-level API. Application authors are encouraged to only subscribe to events that are relevant to their application, which almost never involves low-level networking events.

By keeping the API small, end-user code is also kept small, by allowing simple ideas to be expressed in very few lines of code. Concise expression can be seen in the canonical example of an extremely brief echo server (see Fig. 2), which uses several of the methods and classes described above. While compa-



Notification	Method
Data received from peer.	dataReceived(data)
Connection established with peer.	connectionMade()
Previously-established connection dropped.	connectionLost(reason)

Request	Method Name
Send data.	write(data)
Listen on TCP port, produce a protocol when connections established.	listenTCP(port, factory)
Connect to remote TCP port.	connectTCP(host, port, factory)
Drop a previously-established connection.	loseConnection()
Run the main reactor in an a loop until it is stopped.	run()

Abstract Object	Class Name
Stream-oriented protocol parser.	Protocol
Stream-oriented protocol parser with extensions for POSIX processes.	ProcessProtocol
HTTP request object.	http.Request
Reactor Pattern interface for TCP operations.	IReactorTCP

Figure 1: Examples of Naming

```

from twisted.internet import protocol
from twisted.internet import reactor

# protocol implementation, writes
# what it receives
class Echo(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

# protocol factory, used for state that
# needs to be shared between protocol
# instances. here, that isn't much.
class EchoFactory(protocol.ServerFactory):
    def buildProtocol(self, addr):
        return Echo()

# listen on port 9990 using TCP - listenSSL
# could be used for SSL or TLS
reactor.listenTCP(9990, EchoFactory())
# start the event loop
reactor.run()

```

Figure 2: Echo server

able servers in other frameworks or languages may be as short or as simple, the authors believe that more complex servers will benefit to a larger degree, as in the Conch server (see Section 8).

### 4.3 Make Low-Level Information Available

While Twisted provides a great deal of wrapping to allow portability, certain low-level features are often required. Twisted attempts to provide a multi-level approach to accessing this information. For example, connection failures are described by an exception type. However, sometimes the exact errno is more useful information than the exception type. So, developers may detect, in order of specificity:

- That a connection failed.
- How the connection failed, in a general and portable manner.
- What exact error the operating system reported, if the underlying API provides it.

### 4.4 Similar Events, Different Sources

Run-time polymorphism is a basic feature of object-oriented programming that is too rarely utilized to

provide flexible interfaces.

Twisted takes advantage of Python's dynamic typing to minimize the number of different types of events (method calls) that are necessary. At the lowest level, this allows Twisted to generate `dataReceived` calls from both non-blocking and true asynchronous-style low-level APIs. Similarly, Protocol implementors can ignore most of the distinctions between a TLS and TCP connection, or even a TCP socket and an in-memory class used for testing purposes. All three of these transport types generate `connectionMade`, `dataReceived` and `connectionLost` events, and accept write and `loseConnection` events.

Users of the framework can take advantage of this architecture by creating wrappers that indirect these events by supporting the wrapped object's interface. This can be used to establish bandwidth throttling, connection limiting and other such features that are portable both across any implementation of the reactor object and any kind of protocol.

## 5 Concurrency

### 5.1 Threading

Despite previously-mentioned problems with Python's threading, sometimes using threads is necessary. It can be especially useful when managing threaded C code, since that code need not be affected by the global interpreter lock. An event handler (i.e. a method handling an event) can not run for more than a very short period, since this will stop all service from other clients of the server, freeze the GUI and so on. Since an event handler must not block, blocking or long running operations have to be delegated to threads. Support for threading does have a price, though — without threads there is no real need to make APIs thread-safe. Once threading is introduced, a whole new set of potential problems need to be dealt with to insure such things as race conditions, deadlocks, etc. do not occur.

Twisted provides a set of threading APIs designed to minimize these issues and isolate potential problem spots as much as possible. Operations that need to be run in a thread are added to a queue, which

```
from twisted.internet import reactor

c = 1
# this trivial example function is
# not thread-safe
def increment(x):
    global c
    c = c + x

def threadTask():
    # run non-threadsafe function
    # in thread-safe way -- increment(7)
    # will be called in event loop thread
    reactor.callFromThread(increment, 7)

# add a number of tasks to thread pool
for i in range(10):
    reactor.callInThread(threadTask)
```

Figure 3: Using threads

is then fed to a thread pool. If a thread needs to call a method that is not thread-safe, it can queue the operation via the reactor, and the operation will run in the event loop's thread in the next iteration of the event loop. This gives threads access to all of Twisted's APIs without having to worry about thread-safety issues.

For example, RDBMS access, including Python's generic database access (DB-API), is almost always blocking. Twisted provides a wrapper around DB-API that uses the thread-pool support to allow easy RDBMS use from Twisted applications, hiding the blocking aspect of the API from the user.

### 5.2 Deferreds

In a threading framework, there is only one way to request data: make a function call, get a result as the return value. In an event driven framework, however, functions are divided into two categories; those whose result will be provided immediately as the return value and those whose result will be provided later, as an incoming event.

For example, though a naive programmer may want to structure a request for an HTTP URL as a single blocking call, this would require each HTTP request to spawn a new thread. This approach does not scale beyond a few concurrent requests.

```

# blocking style
try:
    r = blockUntilResult()
except:
    print "An error has occurred"
else:
    print r

# callback style -- notice lack of
# error handling
def getResult(r):
    print r
callWhenResult(getResult)

# Deferred style
def getResult(r):
    print r
def gotError(e):
    print "An error has occurred"
deferred = doSomething()
# gotError will also be called on
# exceptions in getResult:
deferred.addCallback(getResult)
deferred.addErrback(gotError)

```

Figure 4: Example of Deferred vs. other coding styles

### 5.2.1 Don't Call Us, We'll Call You

The standard solution for this issue is to refactor a function so that instead of blocking until data is available, it returns immediately, and the caller passes a callback function that will be called once the data eventually arrives. There are several things missing from this simplistic solution. There is no way to know if the data never comes back; no mechanism for handling exceptions. There is no way to distinguish between different calls to the callback function from different sessions. The Deferred class solves these problems, by creating a single, unified way to deal with deferred results — results that are not immediately available. Deferred encapsulates and extends the concept of a callback.

A Deferred instance is a promise that a function will at some point in the future have a result, and so a Deferred is returned from the function instead of the actual result. Callback functions can be attached to a Deferred object, and once it gets a result these callbacks will be called. In addition Deferreds allow the developer to register a callback for an error, with the default behavior of logging the error.

This is an asynchronous equivalent of the common idiom of blocking until a result is returned or until an exception is raised. As was mentioned previously, multiple callbacks can be added to a Deferred. The first callback in the Deferred's callback chain will be called with the result, the second with the result of the first callback, and so on. Why is this necessary? Consider a Deferred returned by the Twisted RDBMS wrapper - the result of a SQL query. A web widget might add a callback that converts this result into HTML, and pass the Deferred onwards, where the callback will be used by Twisted to return the result to the HTTP client.

The Deferred abstraction is very powerful in that it allows code that looks quite similar to its blocking equivalent, thus making the code easier to understand and maintain. Even more important is the fact that by using the same abstraction for "blocking" results in all parts of the framework, integrating the various systems together is far easier.

For example, methods published via XML-RPC[7] may not be able to calculate the result of the method immediately. Since the XML-RPC framework supports Deferreds, a method published using the XML-RPC framework can simply return a Deferred that will eventually have the result of the method. Since the RDBMS wrapper also returns Deferreds on the result of e.g. a SELECT query, a XML-RPC method can do a SQL SELECT, attach a callback on the resulting Deferred that massages the result and then returns it. The resulting code is almost identical to the equivalent code in a framework that allows blocking.

## 6 Application Facilities

In addition to the lower-level facilities for implementing networked clients and servers, Twisted builds on this functionality to provide higher level libraries required in many applications. This paper will only mention some of the services provided by Twisted, but in addition Twisted provides deployment tools, object/relational mapping, RDBMS event loop integration, directory-based dbm-like storage, bandwidth throttling, utility Python libraries and much more.

Protocol	Client	Server
HTTP	Y	Y
SSH	Y	Y
SMTP	Y	Y
IRC	Y	Y
Telnet	N	Y
SOCKSv4	N	Y
NNTP	Y	Y
FTP	Y	Y
POP3	Y	Y
XML-RPC[7]	Y	Y
SOAP[8]	N	Y
OSCAR (AIM and ICQ)	Y	N
IMAP	Y	Y

Table 1: Protocols implemented by Twisted. Since Twisted is an open source project, protocols are available based on contributions of code from users and developers.

## 6.1 The Middle Level - Protocols

In many cases, the main protocol in a networked application will be a common standardized protocol, not a custom designed one. Email clients use SMTP, POP3 and IMAP, news servers use NNTP and so on. Even if a custom protocol is being used, there is frequently a need to use standard protocols in addition. For example, the application may want to send out an email (and thus require SMTP), or provide a web control-panel (HTTP), or allow remote monitoring with SNMP. Twisted provides implementations of many common protocols, without specifying any policies on usage or backend implementation. This allows developers to easily use common protocols out of the box with Twisted, without having to spend the time developing them from scratch. At the same time, because Twisted exposes the lowest level of the protocol, developers can use the protocols without being restricted by policy decisions which may not fit their specific application.

## 6.2 The High Level - Frameworks

A protocol implementation by itself is not necessarily all that useful. Policies and framework support are needed before a protocol implementation can become a real application. To this end, Twisted includes a number of frameworks providing default policies and support code, implemented on top of

the low-level protocol implementations, although of course developers need not use these policies if they so choose. Some (such as twisted.mail, a SMTP and POP3 server which can be deployed as smarthost and mail drop) are still in a preliminary development state, but others are more extensively developed and mature.

### 6.2.1 twisted.web

twisted.web is a web server framework. It is based on the idea of object publishing: a website is a tree of objects, with the URL corresponding to a specific branch and the resulting objects rendering the result of the HTTP request. twisted.web supports serving static content and CGIs, along with easy integration of objects generating dynamic content. Also supported are XML-RPC and SOAP[8] for simple object publishing (both are simplistic HTTP-based "RPC" protocols).

In addition, twisted.web supports distributed web servers. A HTTP request may be re-sent using Perspective Broker (see 6.3) to another web server that actually handles the request. On UNIX systems, this can be used to allow users to publish homepages with dynamic content (e.g. <http://www.example.com/~itamar/>). Each user runs their own twisted.web server that listens on a Unix socket for PB requests, and the main web server running on port 80 forwards requests for that user to their own web server which is running with the user's security limitations.

Web servers that have been decoupled in this manner provide several advantages. Users can run persistent web services that start up, shut down and restart without affecting the main web server. This approach also simplifies reliably separating privileges between individual users and the web server. The main web server does not need to change user-ID to the user in order to run subprocesses with their security restrictions, so it does not require super-user permissions for anything except the initial binding of port 80.

### 6.2.2 twisted.news

twisted.news is a NNTP server framework. It supports Usenet integration, moderation and other extended NNTP functionality. Storage is designed to



```

# example of testing SOAP server using
# Python interpreter:
#
# $ python soapserver.py &
#
# $ python
# > from SOAP import SOAPProxy
# > url = 'http://localhost:8080/'
# > p = SOAPProxy(url)
# > p.add(1, 2)
# 3
# > p.add(4, 9)
# 13
#
from twisted.web import soap, server
from twisted.internet import reactor

# all methods beginning with 'soap_'
# are published:
class SOAPAdder(soap.SOAPPublisher):
    def soap_add(self, a, b):
        return a + b
site = server.Site(SOAPAdder())
reactor.listenTCP(8080, site)
reactor.run()

```

Figure 5: SOAP server that publishes “add(a, b)” method

be pluggable; currently supported methods include using a RDBMS, and a simple backend for testing purposes that serializes Python objects to disk.

### 6.2.3 twisted.words

twisted.words is a messaging and chat framework, designed to work with multiple protocols. It includes a chat protocol implemented using Perspective Broker (see below) and IRC server support for chat clients. It also has a web based interface allowing users to sign up for accounts. This is an example of Twisted’s power in integrating multiple protocols and services. In addition, the framework provides infrastructure for automated clients, known as bots. One such bot acts as a bridge to channels on other IRC servers, creating fake user objects corresponding to users on the remote server. This is a much more powerful solution than what is possible with most IRC bots, since the bot is integrated into the server, instead of working on the protocol level.

## 6.3 Perspective Broker

Perspective Broker is a remote method invocation framework. It is a message-oriented, asynchronous, authenticated protocol designed for use with multiple languages. In addition to Python implementation included with Twisted, a full implementation exists for Java and support for other languages is underway.

PB was designed from a frustration with other remote object protocols. The central feature that PB supports is the combination of RMI-level flexibility with the connection of untrusted clients. Most RPC or RMI protocols are designed with the assumption that they will be used in a cluster of closely-bound and ultimately-trusted machines (such as CORBA and DCOM); those that are not (such as SOAP and XML-RPC) tend to be limited, extremely verbose, or both. In addition, authentication is an afterthought in many of these protocols. PB uses the simple, flexible and robust capability security model, inspired by the E language[9].

PB is designed to be the “native” protocol of Twisted, and eventually to combine all the abstractions that other protocols in Twisted support. The distributed webserver functionality in Twisted is implemented by representing HTTP protocol abstractions as PB objects and sending them between different Twisted servers. This approach allows greater flexibility to implement servers talking about a “legacy” protocol than the protocol itself would do: for example, PB/HTTP objects may later be extended to include metadata about load-balancing without breaking compatibility.

## 7 End-To-End: An Example Spanning Both High- and Low-Levels

As an example of how applications build upon both low-level and high-level framework components, here is an illustration of a simplified XML-RPC request.

When the reactor receives a new connection event from the operating system, it dispatches to the appropriate instance of a subclass of `Factory` to create a new instance of a subclass of `Protocol`; in this case, the factory is a `twisted.web.server.Site`

```

from twisted.web import server
from twisted.web import xmlrpc

class MyProc(xmlrpc.XMLRPC):
    def xmlrpc_add(self, a, b):
        return a + b

from twisted.internet import reactor
reactor.listenTCP(8080, server.Site(MyProc()))
reactor.run()

```

Figure 6: Creating a simple XMLRPC server.

which creates a `twisted.protocols.http.HTTP` instance.

A TCP Transport, encapsulating the newly-created socket, is attached to the Protocol, and automatically registered with the reactor's event loop. At this point, the reactor awaits read events from the socket. When such events occur, the Transport is notified. The Transport then reads as much data as possible from the socket, and calls the Protocol's `dataReceived` method with that data. This allows the reactor to abstract very different low-level APIs, such as BSD sockets and POSIX AIO, from the Protocol.

At this point, the data is parsed by the Protocol. The HTTP protocol is a subclass of the `LineReceiver` class, which it uses to parse individual lines. `LineReceiver` also supports "raw mode", which enables HTTP to receive lines when parsing headers and raw data when parsing message bodies and/or chunked input.

When a full request is received, a `twisted.web.server.Request` instance is created. At this point, the URL path is used by the `Site` instance to locate a `Resource` instance which represents the HTTP Entity, or "end-point", for the request. In this case, it is a `twisted.web.xmlrpc.XMLRPC` instance.

The XMLRPC instance loads in the XML payload and parses it. Based on the method name within the payload, it calls the appropriate handler method and returns the result.

The result is then generated as a Document Object Model tree, which is serialized to a sequence of bytes. These bytes are written to the `Request` object, which then writes it to the `Transport`. This gets buffered and upon write events on the socket,

```

from twisted.protocols.basic import LineReceiver

class ExampleHTTP(LineReceiver):
    def connectionMade(self, addr):
        self.state = 'command'
        self.req = SimpleHTTPRequest()

    def lineReceived(self, line):
        if self.state == 'command':
            (self.req.cmd, self.req.url,
             self.req.version) = line.split(' ', 2)
            self.state = 'headers'
        elif state == 'headers':
            if line == '':
                if self.req.needsContent():
                    self.setRawMode()
                    self.state = 'command'
            else:
                self.appendHeader(line)

    def rawDataReceived(self, data):
        self.req.bufferContent(data)
        if self.req.contentComplete():
            self.factory.dispatchRequest(self.req)
            self.setLineMode()
            self.req.extraContent()

```

Figure 7: `LineReceiver` example: a sketch of an HTTP Protocol

the Transport will be notified and write out the buffered data to the network.

## 8 Case Study - Conch

Conch is a shell framework, impressive mainly for its implementation of the SECSH protocol (SSHv2). This framework allows users to take advantage of secure shell features in application-level code, as well as providing a full-featured, standalone SSH client and server.

```

class Transport:
    def write(self, data):
        self.buffer += data
        self.activate()

    def readyToWrite(self):
        count = self.skt.send(self.buffer)
        self.buffer = self.buffer[count:]

```

Figure 8: A sketch of a transport's write-buffering.

Conch is a good example of the Twisted design approach providing tangible benefits. Examining some attributes of this system as opposed to other SSH servers, the benefits of a high-level programming language and an asynchronous core will become clearer.

Twisted makes writing servers of this sort easy, since many of the simple features which are considered to be part of implementing a server are either available or unnecessary. For example, in the client, reading the user's password from a terminal is one function already included in Python. This is opposed to OpenSSH's `readpass.h` and `readpass.c`, comprising approximately 150 lines of code. Opening an outgoing TCP connection is implemented in Twisted's core, not Conch; in OpenSSH, there are 2 functions, `ssh_create_socket` and `ssh_connect`, which together are almost 200 lines.

Conch currently provides a large subset of OpenSSH's SSHv2 and is included with the Twisted distribution.

## 8.1 Statistics

While it is quite difficult to empirically substantiate claims like "faster development time" or "reduced code size", some heuristic measurements have been provided which indicate that Conch was implemented quickly, with little cost to efficiency. Since both Conch and all its competitors are ongoing projects with bugfixes happening regularly, it would be difficult to compare development time, but since all implement the same core protocols, implementation complexity can be compared.

Project	SLoC	People
Conch	5,000	1
J2SSH	20,000	7
OpenSSH	64,000	84

Initial benchmarking of Conch shows that it is about the same speed as OpenSSH, though it is faster for certain tasks and slower for others. It can start more sessions per second on the same hardware, but it can transfer fewer encrypted bytes per second (with the Pyco Python accelerator Conch is faster than OpenSSH on throughput and has a much faster connection rate). Considering the vastly simpler implementation and the fact that very little

time has yet been spent on optimization, this is very encouraging. Connections per second and bytes per second were tested over loopback on an AMD Athlon 1800+, with 384MB RAM, running Debian GNU/Linux 3.0 (woody), and using OpenSSH client v3.4p1. OpenSSH did 3 connections/sec and 7.4MB/sec. Conch, with Pyco, did 11 connections/sec and 8.1MB/sec, and without Pyco 8 connections/sec and 3MB/sec.

## 8.2 Features

OpenSSH is not very portable, because it is bound directly to providing UNIX services. Each port is comprised of a separate, subtly different source tree, and all are synchronized against a central repository by using an OpenBSD compatibility layer. It does offer extended functionality such as SSHv1 protocol support. J2SSH is restricted in its functionality because it does not allow the standard operation mode of SSH as a remote shell server (the standard "login to your UNIX account and get a shell prompt" functionality).

Conch has a UNIX-login-savvy server that can replace OpenSSH, "out of the box" (the UNIX specific code in Conch is less than 1000 lines of code out of the total). It is also portable to native Win32, without using UNIX-emulation mechanisms such as Cygwin. The back-end for authentication is easily replaceable, making it feasible to quickly implement an authentication backend that uses Windows authentication information, or a Netinfo database on MacOS X.

Thus, by supporting high-level cross-platform functionality and low-level platform specific features, Conch can function as a replacement for both OpenSSH and J2SSH. It can integrate with the UNIX specific functionality most users expect of an SSH server and client, as does OpenSSH. At the same time, most of the SSH implementation runs on non-UNIX platforms and is easily extendable, as with J2SSH. Thus, one can implement a multi-user terminal based application running inside a single process and accessible via SSH, something which would be vastly more difficult with OpenSSH.

## 9 Related Work

There is a large amount of freely-available networking code on the internet today: so much so that the fundamental programs that drive our networks are the largest credit to the Open Source community. However, few frameworks endeavor to Twisted's scope and generality, and none that the authors are aware of provide as much integrated functionality as it does.

### 9.1 SEDA (Sandstorm)

SEDA is a very similar project to Twisted. As it describes itself:

SEDA is an acronym for staged event-driven architecture, and decomposes a complex, event-driven application into a set of stages connected by queues.[4]

SEDA formally represents the notion of a "Stage" and therefore has more support for gracefully failing in the face of overwhelming load than Twisted does. Many of the abstractions involved are similar.

SEDA does have some multi-protocol support, with applications providing peer-to-peer (Gnutella), email (SMTP) and web (HTTP and HTTPS). While basic protocol functionality is in place, the high-level frameworks to deal with protocol-specific abstractions - similar to twisted.web - are rather light at this point, and Java does not yet provide good standard library support for event-based networking.

SEDA also supports two relatively low-level APIs, the now-standard `java.nio` and their home-grown `nbio`. Sadly, these two APIs do not represent a very abstract way of doing networking. The names of the central abstractions for multiplexing in these packages (`SelectSet` for `nbio`, `Selector` for `java.nio`) imply a certain prejudice in favor of legacy networking APIs. In addition, the choice of Java limits the platform specific functionality available.

Sandstorm or SEDA might be a more appropriate platform for an application which had very serious constraints on its behavior under catastrophic load.

### 9.2 POE

POE is a framework for creating multitasking programs in Perl[6]:

POE parcels out execution time among one or more tasks, called sessions. Sessions multitask through cooperation (at least until Perl's threads become mainstream). [...] POE has become a convenient and quick way to write multitasking network applications.

POE supports many of the same features Twisted does, including GUI event loop integration and support for multiple protocols. POE v0.25 has approximately 28,000 lines of Perl code, but there are many components developed for it that are distributed separately whose equivalent would be distributed with Twisted. POE makes heavy use of the rather verbose Perl object-model, however, and issues with the Perl language are pervasive throughout the framework. The first question in the POE FAQ is illustrative:

The way event handlers receive parameters looks strange at first, but it's a combination of a few common Perl features.

Many of Perl's features create unnecessary problems of communication between modules written by people with differing tastes. Parameter-passing is the most basic feature that an integration system can use. The authors feel that if even something so fundamental will "look strange" to a Perl programmer unfamiliar with POE, this is not an easy-to-use framework.

### 9.3 ACE[5]

The ADAPTIVE Communication Environment (ACE) is a freely available, open-source object-oriented (OO) framework that implements many core patterns for concurrent communication software.

The ACE framework provides a very useful addition to the C++ programmer's toolbelt. However,



it does not alleviate certain basic problems of the C++ language, such as portability issues arising from differing STL implementations, lack of memory protection and build-process integration issues. While it eases the task of writing communications modules, there is little to leverage outside of ACE to enable integration with multiple GUI toolkits or database drivers. ACE has about 480,000 lines of C++ code.

## 10 Summary

Twisted provides a wide range of APIs and functionality for implementing networked software, from low-level event loop support to high-level messaging frameworks. The choice to support such a broad range of functionality in one framework has proven to be the right one.

By supporting both high- and low-level operations, Twisted applications can have the best of both worlds – cross-platform functionality together with platform-specific additional functionality (as in Conch). By using a high-level language, development time can be significantly reduced. By using Twisted's integrated support for multiple protocols, developers can add web-based control panels or email alerts to their server applications or develop more novel programs (one application messages an IRC chat channel and sends out email notifications of commits to a CVS repository.)

Twisted is being used by a broad range of organizations and developers. Twisted is being used by government organizations such as NASA, by companies in the USA, France, Australia and other countries, by open source projects and by individual programmers. It is being used in applications such as batch document processing servers, web servers, chat clients, distributed hash tables, caching proxies, IRC bots, educational software, protocol testing, messaging frameworks and many others[10].

Twisted is available online at:

<http://www.twistedmatrix.com>.

## References

- [1] Kqueue: A generic and scalable event notification facility - Jonathan Lemon
- [2] Java SDK 1.4.1: New IO APIs:  
<http://java.sun.com/j2se/1.4.1/docs/guide/nio/>
- [3] Netcraft Survey:  
<http://www.netcraft.com/survey/>
- [4] SEDA: An Architecture for Well-Conditioned, Scalable Internet Services (2001) - Matt Welsh, David Culler, Eric Brewer
- [5] An Architectural Overview of the ACE Framework (1998) - Douglas C. Schmidt
- [6] POE: a framework for creating multitasking programs in Perl: <http://poe.perl.org>
- [7] XML-RPC: <http://www.xmlrpc.com/>
- [8] SOAP: <http://www.w3.org/TR/SOAP/>
- [9] E Language: <http://www.erights.org/>
- [10] Twisted Success Stories:  
<http://www.twistedmatrix.com/services/success>
- [11] Why Threads Are A Bad Idea (for most purposes) - John Ousterhout:  
<http://www.softpanorama.org/People/Ousterhout/Threads/tsld001.htm>



# In-Place Rsync: File Synchronization for Mobile and Wireless Devices

David Rasch and Randal Burns  
*Department of Computer Science*  
*Johns Hopkins University*  
{rasch,randal}@cs.jhu.edu

## Abstract

The open-source rsync utility reduces the time and bandwidth required to update a file across a network. Rsync uses an interactive protocol that detects changes in a file and sends only the changed data [18, 19]. We have modified rsync so that it operates on space constrained devices. Files on the target host are updated in the same storage the current version of the file occupies. Space-constrained devices cannot use traditional rsync because it requires memory or storage for both the old and new version of the file. Examples include synchronizing files on cellular phones and handheld PCs, which have small memories. The in-place rsync algorithm encodes the compressed representation of a file in a graph, which is then topologically sorted to achieve the in-place property. We compare the performance of in-place rsync to rsync and conclude that in-place rsync degrades performance minimally.

## 1 Introduction

Rsync [18, 19] makes efficient file synchronization a reality. It enables administrators to propagate changes to files or directory trees. To save bandwidth and time, rsync moves a minimum amount of data by identifying common regions between a source and target file. When synchronizing files, rsync sends only the portions of the file that have changed and copies unchanged data from the previous version already on the target. Faster and more efficient methods for synchronizing copies make it easier to manage distributed replicas.

Despite rsync's efficiency, its shortcomings sometimes preclude its use. We address one specific shortcoming. Each time rsync synchronizes a file, it reserves temporary space in which it constructs the new file version. Rsync maintains two copies (one new, one old) on the target for the duration of the transfer. Rsync cannot be used without sufficient temporary space for two copies of a file.

The construction of the new target file in temporary space often renders rsync unusable on mobile devices with limited memory. A popular device by Palm contains only 16MB of memory. For the Palm to keep enough temporary space available might require up to

8MB free (Figure 1). Insufficient space often excludes the Palm from performing traditional rsync and forces a transfer of the entire file. Ironically, handheld systems, compact and convenient machines that can benefit from an efficient propagation of updates, cannot always afford the space overhead of rsync.

Rsync cannot backup or replicate block devices. Although the benefits of compression make rsync well-suited to the task, systems rarely have spare block devices on which to put temporary data.

We have modified rsync so that it performs file synchronization tasks with in-place reconstruction. We call this in-place rsync or *ip-rsync*. Instead of using temporary space, the changes to the target file take place in the space already occupied by the current version. This tool can be used to synchronize devices where space is limited.

In-place reconstruction eliminates the need for additional storage by using the space already occupied by the file [2, 3]. In-place reconstruction seems trivial, but the process must account for hazards that arise when moving a block of data from its original location in the old file to its location in the new file – an operation called a COPY command. Not only does each COPY read a block of the file, but it also overwrites  $k$  bytes. Overwritten regions cannot be used in future COPY commands because they no longer contain the original data.

The goals of the ip-rsync algorithm include: (1) prevent the copying of corrupted data, which has been previously written by another COPY operation; and, (2) minimize compression loss. To prevent the copying of corrupted data, ip-rsync identifies COPY commands that write into regions from which other COPY commands read and then performs the read operation (on the original data) before executing the write. It is not always possible to reorder COPY commands to avoid all conflicts. In this case, ip-rsync discards the conflicting COPY operation. The data corresponding to the COPY are sent from the host to the target. Sending the additional data, instead of copying from the file already on the target, reduces compression and increases the time needed to synchronize files. Ip-rsync implements several heuristics for selecting COPY commands to eliminate that min-

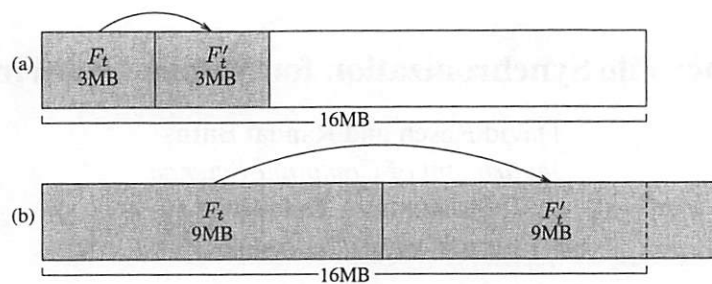


Figure 1: With 16MB of memory: (a) There is enough space for the file  $F_t$  (3MB) and a temporary copy  $F'_t$ . (b) There is insufficient space for a temporary copy  $F'_t$  of  $F_t$  (9MB) and rsync cannot be performed.

imize compression loss.

We describe the in-place rsync utility as an extension to rsync. We start with an overview of the rsync algorithm and a discussion of its performance optimizations. We follow with our algorithm for performing rsync in-place and a discussion of the effect of in-place reconstruction on the algorithm's optimizations.

## 2 Background

Ip-rsync builds on the rsync algorithm for propagating changes between two files located at different sites [18, 19]. Rsync is widely used for backup and restore, as well as file transfer. Rsync finds blocks of data that occur in both the target file and the source file. It saves bandwidth and transfer time when updating the target file by not sending blocks from the source file that exist already in the target file. Rsync operates on a fixed block size in a single pass, or round, over the files. A multi-round version of rsync (*mrsrc*) increases compression by taking multiple passes over files, halving the block size in each subsequent round [9]. Multi-round rsync detects common sections of the files at a fine granularity. However, in multi-round rsync, the sender (source) does not send unmatched data until all rounds are complete. In this way, multi-round rsync loses much of the benefit of the interleaved transfer found in rsync. Rsync transmits unmatched data while searching within the file for matching data. Rsync outperforms *mrsrc* when the similarities and differences between files consist of large sequences. *Mrsrc* performs well when files consist of short matching sequences separated by small differences. *Mrsrc* is more suitable in lower-bandwidth networks in which transfer time dominates. Although our in-place algorithm is suitable for multi-round rsync, we did not implement an ip-*mrsrc* because of *mrsrc*'s limited adoption.

The concepts of rsync influence the design of many distributed systems. In particular, the combination of a weak and strong checksum has been used in a low-bandwidth file system [12], migrating virtual computers [15], and a transactional object store [16].

Rsync has much in common with *delta compression*. Both encode changes between files using COPY and ADD commands. Delta compression differs in its semantics because it compares two files that are collocated, rather than two files separated by a network. Algorithms for delta compression are based on hashing [1, 10, 13] or extensions to Lempel-Ziv compression [4, 6, 8]. The problem of compactly representing versions as a small set of changes was introduced by Tichy as the string-to-string correction problem with block move [17].

In-place reconstruction has been previously addressed for delta compression [2, 3]. The problem and solution are similar to ip-rsync, because they both reorder the execution of commands to avoid conflicting COPY commands.

We feel that in-place reconstruction is more widely applicable in rsync than in delta compression. In-place delta compression can transmit data to a resource limited device. However, it precludes a resource limited sender because it requires both versions of a file to generate a delta encoding. Rsync allows versions to be synchronized between two resource-constrained devices and, therefore, may be used in peer-to-peer and serverless applications.

## 3 Design

Rsync synchronizes two files, bringing an old version of a file on the *target* up to date with a new version of a file on the *source*. Rsync sends as few bytes as possible by detecting common sections of the two versions and using the common data in the old version when building the new version. To detect the common sections, the target generates a weak and strong checksum for blocks in the target file. The target transfers the checksums to the source. On the source, rsync stores the weak checksums in a hash table. The checksums take approximately 100 times less space and bandwidth than the file. The source file is scanned, calculating the weak checksum at each offset. Rsync probes the hash table with each checksum. Upon finding a matching checksum, a strong checksum



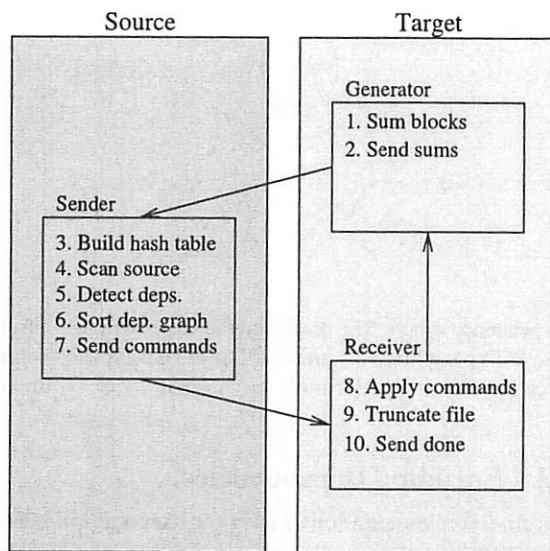


Figure 2: The rsync process triangle with the sequence of steps necessary for an ip-rsync transfer.

is computed and compared. Matching strong checksums indicate matching data with high probability. Rsync encodes matched data as a COPY command. The COPY encodes an action on the target that duplicates data from the reference file in the updated file. The algorithm encodes unmatched data as an ADD command, which includes the data to be added. The target receives encodings from the source. Encodings describe the new file sequentially (from first byte to last) so that the target can reconstruct the new version in a single pass. The use of a weak and strong checksum saves computation by allowing the source to generate and compute a strong checksum only when the weak checksum already matches. Also, the chosen weak checksum saves computation by rolling from offset  $n$  to offset  $n+1$  without recomputing the checksum based on all  $k$  bytes [7]. Rolling checksums observe that strings of length  $k$  at offsets  $n$  and  $n+1$  differ in only two bytes – the first byte of the string starting at  $n$  and the last byte of the string starting at  $n+1$ . The algorithm computes a rolling checksum for offset  $n+1$  by subtracting the contribution of the first byte of the checksum at offset  $n$  and adding the contribution of the last byte of checksum at offset  $n+1$ .

### 3.1 Algorithm

The rsync implementation uses a programming construct referred to as a *process triangle* that defines three processes, one on the source and two on the target (Figure 2). The *generator* process runs on the target and generates the checksums for the target file ( $F_t$ ) that get sent to the *sender* process ( $G \Rightarrow S$ ). The sender scans the source file ( $F_s$ ) for the sums it receives and transmits

the results to the *receiver* process ( $S \Rightarrow R$ ). The receiver applies the delta commands and notifies the *generator* if a file needs to be resent ( $R \Rightarrow G$ ). The *generator* and *receiver* processes run independently on separate files concurrently and proceed independently.

Rsync transmits ADD and COPY commands in the order in which they were detected during a sequential scan of the source file. The target applies the commands in the order received. No destination offsets need to be specified for COPY and ADD commands. The algorithm calculates the destination offset from the previous destination offset and the length of the previous command.

With ip-rsync, the commands undergo a reordering step to facilitate corruption-free, in-place reconstruction. As such, the transmission of the ADD and COPY commands in a non-sequential order requires the explicit specification of a destination offset with each command. The destination offset allows commands to be applied in any order at the target. The extra data in the ip-rsync codeword adds a small overhead to the bandwidth requirements.

Ip-rsync takes the following actions to synchronize a file (the bold text indicate where ip-rsync and rsync differ):

1. *Generator*: Generate weak and strong checksums for each block in the reference target file.
2. *Generator*: Send checksums to the *sender*.
3. *Sender*: Build a hash table from the checksums received.
4. *Sender*: Scan the source file for matches. **Buffer all COPY and ADD commands.**
5. *Sender*: **Construct a dependency graph among COPY commands.**
6. *Sender*: **Topologically sort the dependency graph, breaking cycles as they are detected.**
7. *Sender*: **Send sorted COPY commands, followed by ADD commands, to the receiver.**
8. *Receiver*: Apply commands when received. For each command, **seek to the given offset**, and either copy data or insert the included data.
9. *Receiver*: Truncate the file if it decreased in size.
10. *Receiver*: Alert the *generator* of the file's completion.

In detecting and resolving dependencies, ip-rsync sacrifices some concurrency. The sender conducts an analysis of all COPY commands prior to transmitting any commands to the receiver. This causes ip-rsync to wait until all COPY commands are found and analyzed before transmitting data. Traditional rsync overlaps detecting and transmitting matches by sending data to the

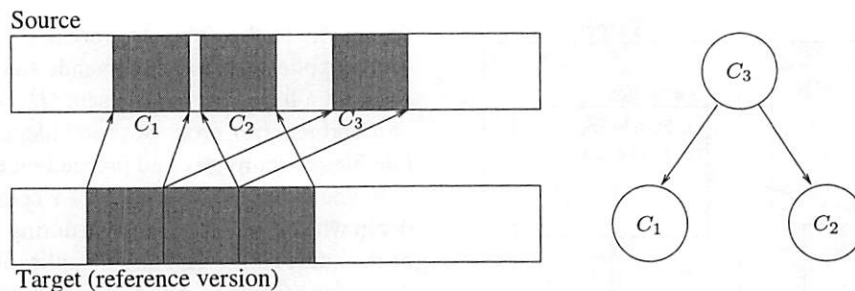


Figure 3: An example of a file synchronization and the associated dependency graph. The blocks on the target will move from their original location (bottom) to match their location on the source (top). To paraphrase the graph,  $C_3$  must be completed before  $C_1$  or  $C_2$  since the destination of these blocks will corrupt the source for  $C_3$ ; *i.e.*, perform  $C_3$  first so that  $C_1$  or  $C_2$  do not overwrite its source data.

receiver immediately. The sender detects dependencies among all COPY commands and waits for the discovery of all commands before reordering.

### 3.2 A Simple Example

We now digress to a simple, yet flawed algorithm, which one might propose as an alternative to the in-place rsync algorithm. The following algorithm is a trivial modification to rsync. Its flaw results in significant compression loss. This example motivates the need for the complexity (buffering and dependency detection) of our solution. The simple algorithm runs rsync and employs the following heuristic: if any COPY command refers to a block location which precedes the current write offset, then that block has been overwritten with new data and can no longer be used for COPY commands. The lost data is resent over the network link as an ADD command. The advantage of this algorithm is the similarity to the original algorithm and its ability to overlap I/O with the computation of checksums.

This simple algorithm is sensitive to changes which insert data into the source file. Such changes cause the simple algorithm to generate many ADD commands, resulting in large compression losses. Consider synchronizing a pair of files that are identical except that the source file has some data (as few as 1 or 2 bytes) inserted at the beginning. These few bytes prevent any data from being copied.

Reordering commands is necessary in spite of the rarity of cycles in dependency graphs. To evaluate the value of ordering, we look at compression in the naive algorithm. Run on our data set, the simple algorithm uses ADD commands to replace 52% of the COPY commands. Each ADD sends  $k$  extra bytes over the network link. This naive algorithm cuts compression in half when compared with traditional rsync. Our final algorithm's complexity proves necessary to avoid a drastic loss in compression.

### 3.3 Encoding Dependencies

Our final implementation of ip-rsync uses a graph-based algorithm that detects and resolves dependencies, preventing corruption of the target file. This comes at the cost of a delay to generate and process the conflicts, losing some of the benefit of overlapping network transfer with checksum generation.

In ip-rsync, the generator (steps 1 and 2) mirrors the corresponding actions of traditional rsync. The process which builds the hash table in step 3 also remains unmodified. However, during the scan in step 4, ip-rsync's sender buffers ADD and COPY commands in memory, instead of sending them immediately. Buffering all commands allows ip-rsync to detect dependencies and reorder commands prior to sending data.

Buffering COPY and ADD commands consumes much less space than traditional rsync requires. For a COPY command, ip-rsync needs space to store the source block and the target offset only. ADD commands require an extra field for the length of the data. The algorithm does not store the raw data for ADD commands in memory. Rather, ip-rsync stores the meta-data in memory and reads data directly from the source file when sending an ADD command.

The algorithm constructs a directed graph in which edges represent ordering constraints among commands (Figure 3). A topological sort of the graph determines an execution order for processing COPY commands on the target. When a total topological ordering proves impossible because of cycles in the graph, ip-rsync converts nodes that copy data in the file to commands that add the data explicitly. This results in lost compression. We later describe several heuristics for breaking cycles.

The source puts buffered nodes into a structure that points to the COPY command and contains fields necessary for dependency detection, topological sorting, transmission to the receiver, and deallocation. The fields of the graph node are a pointer to the COPY command that the node represents, a "visited" field used to encode

**Algorithm 3.1:** DFS(*graph*)

```

procedure VISIT(node)
  if not node.VISITED
  then { node.VISITED  $\leftarrow$  true
        for each edge  $\leftarrow$  EDGES(node)
        do VISIT(TARGET(edge))

main
  for each node  $\leftarrow$  NODES(graph)
  do VISIT(node)

```

Figure 4: Pseudo-code for depth-first search. Initially nodes are in the *UNVISITED* state.

**Algorithm 3.2:** MODIFIED-DFS(*graph*)

```

procedure VISIT(node)
  if node.STACK
  then { node.DELETED  $\leftarrow$  true
        DELETE(node)
  if not node.VISITED
  then { node.STACK  $\leftarrow$  true
        for each edge  $\leftarrow$  EDGES(node)
        do VISIT(TARGET(edge))
        node.VISITED  $\leftarrow$  true

main
  for each node  $\leftarrow$  NODES(graph)
  do VISIT(node)

```

Figure 5: Modified depth-first search.

states during topological sort, a reference counter for deallocation, a pointer to the next node in topological order (initially NULL), and finally a pointer to the first in a linked list of outgoing edges.

The sender also buffers ADD commands. ADD commands are self-describing and, therefore, require no data from the old version. As a result, ADD commands need no reordering.

After all commands have been buffered, the COPY graph is passed to a dependency-detection function (step 5). This function detects all dependencies with an  $O(n \lg n)$  algorithm, where  $n$  is equal to the number of bytes in the larger of the source file or the target file [2]. Each generated edge has two fields: the target node, and a pointer to the next outgoing edge from the source. With the full graph constructed, the sender topologically sorts the nodes using a modified depth first search (DFS) algorithm.

The algorithm used to topologically sort the graph in step 6 modifies DFS to make it operate on graphs that contain cycles. We present pseudo-code for depth-first

search (Figure 4) and the modified depth-first search used in our algorithm (Figure 5). DFS (for acyclic graphs) is a “stack” algorithm, pushing nodes onto a stack as they are visited. In DFS, nodes take on two states – *UNVISITED* and *VISITED*. A node remains *UNVISITED* until the algorithm pops it off the *STACK* during its traversal. The algorithm marks the completed node *VISITED*. Modified-DFS for topological sort uses an additional *STACK* state to record the order in which the algorithm visits each node. The *STACK* state helps detect and break cycles. Modified-DFS also adds a *DELETED* state to encode nodes that have been deleted. Just as in DFS, each node begins *UNVISITED*. Analogous to DFS, the algorithm calls *Visit* on each node. *Visiting* a node places it on the stack and marks it in the *STACK* state. Subsequently, the algorithm *Visits* any neighbors of the node that are not *VISITED* or *DELETED*. When done with the neighbors, the algorithm removes the node from the stack, marks the node *VISITED* and places it on the front of an output list. If the *Visit* procedure finds a neighbor already marked *STACK*, a cycle exists in the graph. The algorithm marks this node *DELETED*, which breaks the cycle. A compensating ADD command is created in place of the *DELETED* COPY. The topological sort algorithm is  $O(V + E)$  as it examines each vertex at least once and traverses every edge. We will examine alternative metrics and procedures for resolving cycles in the next section.

Ip-rsync sends the COPY commands to the target in topologically sorted order. Upon sending a COPY command, it deallocates the corresponding node along with its remaining edges. After sending all COPY commands, ip-rsync sends ADD commands according to the ADD list, including the ADD commands that correspond to deleted COPY commands (step 7). If necessary, the target truncates the file to the new size and the synchronization is complete.

### 3.4 Cycle Breaking

Any cycles found in the dependency graph represent a set of COPY commands that mutually depend on each others’ completion. The only ways to find a valid ordering of the COPY commands in a cycle are to remove an edge of the cycle or remove a node (COPY command) entirely.

The cycle breaking method described in the previous section deletes an entire node to resolve a cyclic dependency. To compensate, the sender sends an ADD command with the data that should have been copied by the original command. This deletion costs  $k$  bytes of compression. The sender marks the node as deleted and the process continues. In-place delta compression [2] uses



another metric that finds and deletes the smallest COPY command in a cycle in order to minimize compression loss. Ip-rsync need not distinguish between nodes based on size. Rsync has a fixed block size; all COPY commands have the same length.

We implement an alternative method for breaking cycles. This “trimming” method removes an edge of the cycle, rather than a node. An edge occurs when the read region of one COPY overlaps the write region of another COPY. The edge is eliminated by shrinking one COPY command, so that it no longer overlaps with the other COPY. An ADD command is generated that covers the trimmed region. We call this trimming a node, because it reduces the read and write regions described by the node. Trimming preserves some of the benefit of existing COPY commands when breaking cycles. When ip-rsync with trimming detects a cycle, it scans through all nodes in that cycle. The scan examines the overlap between each pair of nodes. It trims the dependency with the least overlap. The goal of this policy is to minimize compression loss. After trimming a node, ip-rsync checks existing edges pointing to the trimmed node to ensure that dependencies have not changed. It is possible that the trimmed node no longer conflicts (overlaps) with other nodes in its edge list.

To preserve the format and encoding of COPY commands, the algorithm does not change a COPY command when trimming the corresponding graph node. Instead, the algorithm allows the COPY command to write corrupt data and repairs the corrupt bytes with an ADD command. This preserves the fixed-size block used by rsync. During the transmission phase, no changes are made to the protocol and the target copies the full block for a trimmed node, which includes corrupted bytes in the overlapping region. The ADD command generated when trimming the node repairs the corrupted data.

## 4 Performance

Our experimental results compare ip-rsync to traditional rsync and evaluate different policies for breaking cycles found in ip-rsync graphs. Results indicate that ip-rsync degrades performance in bandwidth constrained environments, which we expect since it always transmits more data over the network. However, when factors other than bandwidth limit performance, our tests show that in-place rsync outperforms traditional rsync. Although ip-rsync requires extra computation, it reduces disk writes and file-system block allocations.

Our experimental data set provides an example of files used on handhelds that need to be updated over wireless networks. The data set consists of 1523 pairs of versioned files obtained from <http://www.handhelds.org/>. The files include

a variety of kernel binaries and compiled programs that are intended to be downloaded to handheld computers. The files in our dataset target the same audience and devices that ip-rsync benefits the most. To collect data, we downloaded the software archive and ran scripts that search the archive for multiple versions of the same files. The original and processed data are available from the Hopkins Storage Systems lab at <http://hssl.cs.jhu.edu/ipdata/>.

In our experiments, we synchronize each pair of versions with rsync and in-place rsync. For in-place rsync, we compare two different cycle breaking methods: “delete node” that deletes the final node found in a cycle and “trim node” that trims the least possible number of bytes in each detected cycle.

Ip-rsync incurs some compression loss from encoding overhead. In-place rsync adds four bytes to each twelve byte codeword in order to encode offsets in the target file. These four bytes have a very different effect on compression and on bandwidth overhead. We illustrate this point with a simple worst-case example in which all commands are COPY commands. This results in negligible compression loss: 4 bytes degrade compression by 0.55% in the default 700 byte block. However, the 4 bytes increase the bandwidth by 33% – 16 bytes per COPY codeword as opposed to 12 bytes. The overall bandwidth overhead of in-place rsync in our experiments is less than 5%, much less than the 33% worst case bound. Data transferred in ADD commands dominates bandwidth, which mitigates overhead from codewords.

Overall, ip-rsync achieves compression almost identical to rsync. In addition to encoding overheads, ip-rsync loses compression when eliminating cyclic dependencies. With the delete-node policy, the compression lost by ip-rsync compared to rsync was 0.543%. The trim-node policy cost 0.545% in compression. The difference is negligible. The overall increase in transmitted data averages 0.544% of the size of the original file.

In-place rsync pays for its decreased parallelism and compression with longer transfer times at low bandwidths. For low bandwidths, the in-place algorithm spends 10% more time in the scanning and command transmission steps combined than the original rsync algorithm requires to complete the parallel hash search and command transmission phase. The extra time spent in these steps is fundamental to algorithms that reorder commands and is therefore unavoidable.

The performance of ip-rsync scales almost identically to that of rsync (Figure 6). Only at the smallest bandwidths can the effect of transferring the offset with each command be seen. Otherwise, ip-rsync has negligible latency and bandwidth overhead.



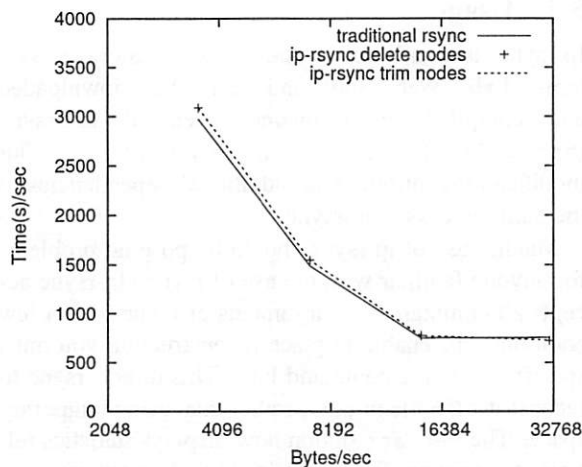


Figure 6: The graph shows transfer times of the entire dataset at different bandwidths using rsync and ip-rsync with two cycle breaking heuristics. Ip-rsync performs more slowly for bandwidth-limited transfers. Rsync incurs extra overhead in the *ext2* file system when allocating space in the temporary file and, as a result, ip-rsync outperforms rsync at higher bandwidths. The cycle breaking strategies make no noticeable difference in transfer time.

Although ip-rsync loses parallelism within a single file, it preserves parallelism across many files. Ip-rsync (and rsync) overlap the transmission of one file with scanning in a subsequent file. Our results show that overlapped execution is the dominant form of parallelism. In one instance of a bandwidth-limited transfer, the entire ip-rsync synchronization took 300 seconds and rsync required 290 seconds. The ip-rsync algorithm transmitted an extra 312,304 bytes, which accounts for the increase in end-to-end time. The standard deviation of the compression loss per file (the average compression loss of is 0.544%) was only 0.8%.

The overhead that arises from decreased parallelism is negligible. Comparing rsync and ip-rsync on a single file would indicate a greater performance loss. However, parallelism lost within a single file is recovered when overlapping multiple files.

While developing ip-rsync, we expected that rsync would outperform ip-rsync in all cases. We believed this because the changes to the algorithm include no improvements in bandwidth, memory usage, or processing. Also, rsync allows for more parallelism between the source and target.

When I/O limits performance (as opposed to bandwidth), ip-rsync outperforms rsync by 2-3% (see Figure 6 at bandwidths greater than 20,000). We account for the decrease in speed by considering the areas where ip-rsync reduces overhead. Ip-rsync performs fewer file-system block allocations using FFS-like file systems [11] that update data in-place. Our tests used the

*ext2* file system. However, a copy-on-write file system [5, 14] would negate these benefits, as every write allocates a new file system block. In our further discussion, we refer to our trials on *ext2*. By eliminating the need for temporary space, ip-rsync allocates space only as required to increase the file size. In contrast, rsync allocates all blocks in the temporary file, then deallocates the original blocks. The time required to allocate space for a temporary copy for a traditional rsync balances out the decreased parallelism and compression. Furthermore, ip-rsync requires fewer disk writes when files are changed slightly. Ip-rsync can ignore COPY commands which have the same source and destination. Traditional rsync must copy all blocks regardless of whether they remain at the same offset in the file.

Buffering ADD and COPY commands for in-place rsync requires extra memory, but utilizes far less space than that required by rsync. The amount of extra memory scales with file size, but is much smaller than that of a temporary copy. On average, the extra memory needed is only 3.1% of file size, with a standard deviation of 2.9%. The method for cycle breaking by trimming nodes required 3.2% of the file size in data memory, while the delete node algorithm required only 3.0%. The buffering of in-place rsync can be problematic in resource-limited environments. Although we attempt to minimize the space required, it is possible that the algorithm can exceed available memory. We plan to implement the windowing techniques described in section 6.1 to address this problem.

## 5 Implementation

Adding in-place reconstruction to rsync changes the operation and the usage of the rsync utility in minor ways. In-place reconstruction affects interfaces, error handling, and the information and statistics that rsync generates.

Because ip-rsync updates a single copy of the data on the target, it changes the operational semantics of the tool, particularly when errors occur. Modifications to the file at the target cannot be isolated from a process reading data concurrently. New failure and recovery scenarios occur; incomplete synchronizations leave a partially updated file on the target that cannot be recovered to either the old or the new version.

### 5.1 Error Handling

Ip-rsync loses the atomic update property of rsync. Rsync creates a temporary file that contains the updated version. When the update completes, rsync calls `rename()` which unlinks and atomically replaces the existing version of the file. Processes with an open handle to the old file continue to read the old data until they

reopen the file. New processes open the new version of the file. When operating on a single version, ip-rsync makes many intermediate changes to a file. The old version of the data is irrecoverably modified. Even if ip-rsync completes successfully, inconsistent views of data may occur during its operation; applications reading the file concurrently with an update by ip-rsync may read from both the old and new version.

To avoid these inconsistent views, ip-rsync opens files exclusively (rsync opens files in shared mode by default). If another process/application has the file open, ip-rsync fails, leaving the original file intact. If another process attempts to open the file during an ip-rsync session, the process either fails to open the file or blocks awaiting ip-rsync's completion. The outcome depends on the arguments to `open()` and operating system semantics.

If a failure occurs during synchronization, ip-rsync may leave the target file in an inconsistent state. This occurs when the network, receiver process or sender process fails after the receiver has written data. If the receiver process fails, no recovery action can be taken. Rsync leaves a temporary file in the file system and ip-rsync leaves an incompletely synchronized file. The inconsistent file left by ip-rsync does not present a problem upon restart; the source contains the new version of the data which is synchronized in a new ip-rsync session against the inconsistent data. If the sender or network fails, the receiver process continues to run and may take recovery action. Rsync merely removes the temporary file, preserving the state of the file prior to synchronization. Ip-rsync cannot recover the original state.

We balance several factors when deciding how ip-rsync should handle inconsistent files. Options include: (1) deleting the file at the receiver and (2) leaving the inconsistent file in the file system. The first approach prevents the application from reading inconsistent data, but discards a file that contains data that makes a subsequent rsync complete quickly. The second approach has the opposite properties, allowing inconsistent data to be read, but preserving the file for faster synchronization. We realize both benefits by having ip-rsync rename the corrupt file, creating a hidden recovery file that contains the inconsistent data. The original file is effectively deleted so that applications cannot access inconsistent data under the old file name. However, the recovery file is available to be used in a subsequent rsync session. When ip-rsync uses the recovery file, it updates the recovery file in-place and then renames the hidden file to the original file name. Renaming the recovery file avoids producing two copies of the file, which might exceed to storage capacity of the target. The recovery file is not implemented in our current release.

## 5.2 Usage

Ip-rsync is available at the Hopkins Storage Systems Lab Web site and can be downloaded and compiled in a manner identical to rsync (<http://hssl.cs.jhu.edu/iprsync/>). Our modifications introduce no additional dependencies to the build process of ip-rsync.

Making use of ip-rsync should be pose no problems for anyone familiar with the use of rsync. Ip-rsync accepts all command-line arguments of rsync with a few additions. To enable in-place reconstruction you must specify `-i` on the command line. This directs rsync to reconstruct the file in-place rather than using temporary space. The `--stats` option now displays statistics relevant to ip-rsync. The statistics include memory overhead, bandwidth overhead, and compression loss due to in-place reconstruction.

To synchronize a file using in-place reconstruction across the network, a user invokes rsync with the source file and the destination file: `rsync -i source.txt host.example.com:/path/to/dest.txt`. A few messages will appear noting the progress of the synchronization and indicating its successful completion.

Ip-rsync's in-place reconstruction is not compatible with previous versions of rsync. Thus, both hosts involved in the transfer must support in-place reconstruction. Ip-rsync maintains backward compatibility and will synchronize with a peer running rsync.

## 6 Future Work

Ip-rsync requires more detailed experiments in order to quantify tradeoffs between compression and parallelism and to identify further opportunities for optimization. Rsync itself is highly optimized for parallel execution within a single file and between multiple files [19]. Rsync is widely used because it works so well. We intend to follow this example. Although we have identified unfinished work items (in windowing and error recovery), the most important future work will feed experimental results back into the design of ip-rsync.

Even though we designed ip-rsync for mobile and wireless devices, our experiments reveal that ip-rsync works well in higher-bandwidth networks as well. They show that for bandwidths greater than 20,000 KB/sec, ip-rsync actually outperforms rsync. This result implies that ip-rsync would perform well for synchronizing block devices. In-place reconstruction is necessary because block devices are large (much larger than memory) and there is not spare capacity to write a temporary copy of a whole block device. Experiments need to be conducted to validate these claims.

Reduced concurrency within a single file degrades performance in ip-rsync. In our current experiments,

much of this loss is regained through the semi-parallel transfer of multiple files; sending the data of a file while scanning for matching data in another file. However, rsync is commonly used to synchronize single files. More detailed experiments that examine performance when transferring single files will isolate performance losses from reduced parallelism, and indicate ip-rsync's suitability for single file transfer. Such experiments are essential to understand the benefit of reduced I/O in ip-rsync. When combined with semi-parallel execution, the gains of reducing I/O can make ip-rsync outperform rsync. Experiments on single files would allow us to measure the relative value of reduced I/O when compared with semi-parallel transfer.

## 6.1 Windowing

Ip-rsync runs out of space when the size of the buffered commands exceed the available space. Although buffered commands are much smaller than the original data, the algorithm must gracefully handle cases in which the buffered commands exhaust the available storage. This problem arises frequently when synchronizing block storage devices that may be orders of magnitude larger than a system's memory.

There is no trivial way to address buffer overflow by pruning the buffered commands and the graph they induce, nor by completing some commands early. Ip-rsync must retain all commands until the encoding is complete. Otherwise, if the algorithm were to discard a command, then dependencies between the discarded and subsequent commands remain undetected. Undetected dependencies cause an incomplete ordering and corrupt data.

We have designed, but not implemented, an *overlapped windowing* technique to address buffer overflow that organizes a file into multiple regions each of which are processed independently. It is an on-line process in which variable sized independent windows are created as ip-rsync approaches its memory limit. The read regions of the windows may overlap, but the write regions are disjoint.

During synchronization, the ip-rsync sender encodes data in an active window, which it uses until it exhausts memory. The active window has a start offset and goes to the highest byte in the file. When ip-rsync does not reach the memory limit, synchronization completes in a single window. When the memory limit is reached, the sender stops encoding data and completes processing on the buffered commands, *i.e.*, commands are topologically sorted and sent to the receiver where they are applied. The buffered commands are discarded and processing begins in the next window, starting at the first un-encoded byte.

Ip-rsync defines independent windows based on read/write dependency information. The algorithm encodes commands that read data in the active window only.

Windowing can be expressed equivalently as rewriting the dependency graph used by ip-rsync. When the graph becomes too large, the buffered graph is rewritten as a single node that writes data to the region starting at offset 0 and ending at the highest encoded offset. Furthermore, the algorithm cannot reorder this rewritten node with respect to the subsequent commands, because the command that the node represents has already been completed.

The implementation of windowing offers an opportunity to further parallelize ip-rsync. Our current windowing design (described above) minimizes compression loss by transferring as much of the file as possible in a single window; frequently this means the whole file when memory is not exhausted. We have identified two alternative windowing designs that increase parallelism in exchange for compression. One design partitions the files into fixed-size windows before scanning. Then, ip-rsync operates on each window independently, treating each window as a separate pair of files and overlapping transfer between windows. This approach does degrade compression, because blocks that match in different windows cannot be encoded. Another design for large files implements two active windows that are sized dynamically. One at the front of the file and one at the end of the file. This allows two processes to execute concurrently on the same file. When the algorithm exhausts memory, one window (or both windows) are completed to reclaim space. The algorithm completes when the windows collide in the middle of the file. An experimental comparison of different windowing policies will help us understand compression versus parallelism tradeoffs in large files.

Overlapped windowing is somewhat similar to the windowing technique used in delta compression [8], in which files are partitioned into non-overlapping regions. Delta compression defines codeword formats for windows, but does not specify how they are constructed or evaluated. Overlapping windowing differs from the rolling window techniques commonly used in data compression. The "Lempel-Ziv" family of algorithms uses a fixed-sized buffer, discarding the oldest strings, in order to bound the amount of space used by the string library. Rolling windows create a compression versus memory tradeoff — there are no correctness implications and the file is not "partitioned".



## 7 Conclusions

We have described the design, implementation, and performance of in-place rsync for synchronizing files among mobile and wireless devices. Ip-rsync is a modification to the open-source rsync utility so that files may be updated in-place: in the memory or storage that the current version occupies. The algorithms of ip-rsync use a graphical representation of the operations in an rsync encoding to detect when in-place updates would corrupt data, and then topologically sorts these operations to avoid such conflicts. When compared with rsync, ip-rsync loses 0.5% compression from encoding overheads and breaking cyclic dependencies. Ip-rsync increases transfer time in bandwidth-constrained environments, but can increase performance in I/O constrained environments by avoiding the creation of a temporary file.

Although in-place reconstruction can degrade both compression and transfer time, it makes file synchronization available in space-constrained environments where rsync alone does not function. The benefits of file synchronization can be brought to mobile and wireless devices in exchange for minor performance losses.

## References

- [1] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured input with differential compression. *Journal of the ACM*, 49(3), 2002.
- [2] R. Burns and D. D. E. Long. In-place reconstruction of delta compressed files. In *Proceedings of the Symposium on Principles of Distributed Computing*, 1998.
- [3] R. Burns, L. Stockmeyer, and D. D. E. Long. In-place reconstruction of version differences. *IEEE Transactions on Knowledge and Data Engineering (to appear)*, 2003.
- [4] M. C. Chan and T. Y. C. Woo. Cache-based compaction: A new technique for optimizing web transfer. In *Proceedings of the IEEE INFOCOM Conference*, 1999.
- [5] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter Conference*, 1994.
- [6] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An empirical study of delta algorithms. In *Proceedings of the 6th Workshop on Software Configuration Management*, March 1996.
- [7] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [8] D. G. Korn and K.-P. Vo. Engineering a differencing and compression data format. In *Proceedings of the USENIX Annual Technical Conference*, 2002.
- [9] J. Langford. Multiround Rsync. Technical Report Available at [www.cs.cmu.edu/~jcl/research/mrsync/mrsync.ps](http://www.cs.cmu.edu/~jcl/research/mrsync/mrsync.ps), Dept. of Computer Science, Carnegie-Mellon University, 2001.
- [10] J. MacDonald. Versioned file archiving, compression, and distribution. Technical Report Available at <http://www.cs.berkeley.edu/~jmacd/>, UC Berkeley, 2000.
- [11] M. K. McKusick, W. N. Joy, J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3), 1984.
- [12] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the Symposium on Operating Systems Principles*, 2001.
- [13] C. Reichenberger. Delta storage for arbitrary non-text files. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, 1991.
- [14] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), 1992.
- [15] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.
- [16] Z. H. Stephen, S. M. Blackburn, L. Kirby, and J. Zigman. Platypus: Design and implementation of a flexible high performance object store. In *Proceedings of the 9th International Workshop on Persistent Object Systems*, 2000.
- [17] W. F. Tichy. The string-to-string correction problem with block move. *ACM Transactions on Computer Systems*, 2(4), November 1984.
- [18] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.
- [19] A. Tridgell and P. Mackerras. The Rsync algorithm. Technical Report Available at [http://samba.anu.edu.au/rsync/tech\\_report/tech\\_report.html](http://samba.anu.edu.au/rsync/tech_report/tech_report.html), Australian National University, 1998.



# NFS Tricks and Benchmarking Traps

Daniel Ellard, Margo Seltzer  
Harvard University  
{ellard,margo}@eecs.harvard.edu

## Abstract

We describe two modifications to the FreeBSD 4.6 NFS server to increase read throughput by improving the read-ahead heuristic to deal with reordered requests and stride access patterns. We show that for some stride access patterns, our new heuristics improve end-to-end NFS throughput by nearly a factor of two. We also show that benchmarking and experimenting with changes to an NFS server can be a subtle and challenging task, and that it is often difficult to distinguish the impact of a new algorithm or heuristic from the quirks of the underlying software and hardware with which they interact. We discuss these quirks and their potential effects.

## 1 Introduction

Despite many innovations, file system performance is steadily losing ground relative to CPU, memory, and even network performance. This is due primarily to the improvement rates of the underlying hardware. CPU speed and memory density typically double every 18 months, while similar improvements in disk latency have taken the better part of a decade. Disks do keep pace in terms of total storage capacity, and to a lesser extent in total bandwidth, but disk latency has become the primary impediment to total system performance.

To avoid paying the full cost of disk latency, modern file systems leverage the relatively high bandwidth of the disk to perform long sequential operations asynchronously and amortize the cost of these operations over the set of synchronous operations that would otherwise be necessary. For write operations, some techniques for doing this are log-structured file systems [18], journalling, and soft updates [21]. For reading, the primary mechanism is *read-ahead* or *prefetching*. When the file system detects that a process is reading blocks from a file in a predictable pattern, it may optimistically read blocks that it anticipates will be requested soon. If the blocks are arranged sequentially on disk, then these “extra” reads can be performed relatively efficiently because the incremental cost of reading additional contiguous

blocks is small. This technique can be beneficial even when the disk blocks are not adjacent, as shown by Shriver *et al.* [23].

Although there has been research in detecting and exploiting arbitrary access patterns, most file systems do not attempt to recognize or handle anything more complex than simple sequential access – but because sequential access is the common case, this is quite effective for most workloads. The Fast File System (FFS) was the pioneering implementation of these ideas on UNIX [12]. FFS assumes that most file access patterns are sequential, and therefore attempts to arrange files on disk in such a way that they can be read via a relatively small number of large reads, instead of block by block. When reading, it estimates the sequentiality of the access pattern and, if the pattern appears to be sequential, performs read-ahead so that subsequent reads can be serviced from the buffer cache instead of from disk.

In an earlier study of NFS traffic, we noted that many NFS requests arrive at the server in a different order than originally intended by the client [8]. In the case of read requests, this means that the sequentiality metric used by FFS is undermined; read-ahead can be disabled by a small percentage of out-of-order requests, even when the overall access pattern is overwhelmingly sequential. We devised two sequentiality metrics that are resistant to small perturbations in the request order. The first is a general method and is described in our earlier study. The second, which we call *SlowDown*, is a simplification of the more general method that makes use of the existing FFS sequentiality metric and read-ahead code as the basis for its implementation. We define and benchmark this method in Section 6.

The fact that the computation of the sequentiality metric is isolated from the rest of the code in the FreeBSD NFS server implementation provides an interesting testbed for experiments in new methods to detect access patterns. Using this testbed, we demonstrate a new algorithm for detecting sequential subcomponents in a simple class of regular but non-sequential read access patterns. Such access patterns arise when there is more than one reader concurrently reading a file, or when there is one reader accessing the file in a “stride” read

pattern. Our algorithm is described and benchmarked in Section 7.

Despite the evidence from our analysis of several long-term NFS traces that these methods would enhance read performance, the actual benefit of these new algorithms proved quite difficult to quantify. In our efforts to measure accurately the impact of our changes to the system, we discovered several other phenomena that interacted with the performance of the disk and file system in ways that had far more impact on the overall performance of the system than our improvements. The majority of this paper is devoted to discussing these effects and how to control for them. In truth, we feel that aspects of this discussion will be more interesting and useful to our audience than the description of our changes to the NFS server.

Note that when we refer to NFS, we are referring only to versions 2 (RFC 1094) and 3 (RFC 1813) of the NFS protocol. We do not discuss NFS version 4 (RFC 3010) in this paper, although we believe that its performance will be influenced by many of the same issues.

The rest of this paper is organized as follows: In section 2 we discuss related work, and in Section 3, we give an overview of file system benchmarking. We describe our benchmark and our testbed in Section 4. In Section 5, we discuss some of the properties of modern disks, disk scheduling algorithms, and network transport protocols that can disrupt NFS benchmarks. We return to the topic of optimizing NFS read performance via improved read-ahead, and define and benchmark the *SlowDown* heuristic in Section 6. Section 7 gives a new cursor-based method for improving the performance of stride access patterns and measures its effectiveness. In Section 8, we discuss plans for future work and then conclude in Section 9.

## 2 Related Work

NFS is ubiquitous in the UNIX world, and therefore has been the subject of much research.

Dube *et al.* discuss the problems with NFS over wireless networks, which typically suffer from packet loss and reordering at much higher rates than our switched Ethernet testbed [6]. We believe that our *SlowDown* heuristic would be effective in this environment.

Much research on increasing the read performance of NFS has centered on increasing the effectiveness of client-side caching. Dahlin *et al.* provide a survey of several approaches to cooperative client-side caching and analyze their benefits and tradeoffs [4]. The NQNFS (Not Quite NFS) system grants short-term leases for cached objects, which increases performance by reduc-

ing both the quantity of data copied from the server to the client and the number of NFS calls that the client makes to check the consistency of their cached copies [10]. Unfortunately, adding such constructs to NFS version 2 or NFS version 3 requires non-standard changes to the protocol. The NFS version 4 protocol does include a standard protocol for read leases, but has not achieved widespread deployment.

Another approach to optimizing NFS read performance is reducing the number of times the data buffers are copied. The zero-copy NFS server shows that this technique can double the effective read throughput for data blocks that are already in the server cache [11].

In contrast to most previous work in NFS read performance, we focus on improving read performance for uncached data that must be fetched from disk. In this sense, our work is more closely related to the studies of read-ahead [23] and the heuristics used by FFS [13].

## 3 Benchmarking File Systems and I/O Performance

There has been much work in the development of accurate workload and micro benchmarks for file systems [22, 24]. There exists a bewildering variety of benchmarks, and there have been calls for still more [16, 19, 27].

Nearly all benchmarks can be loosely grouped into one of two categories: *micro* benchmarks, such as *lm-bench* [14] or *bonnie* [1], which measure specific low-level aspects of system performance such as the time required to execute a particular system call, and *macro* or *workload* benchmarks, such as the SPEC SFS [26] or the Andrew [9] benchmarks, which estimate the performance of the system running a particular workload. Workload benchmarks are used more frequently than micro benchmarks in the OS research literature because there are several that have become “standard” and have been used in many analyses, and therefore provide a convenient baseline with which to compare contemporary systems and new techniques. There are two principal drawbacks to this approach, however – first, the benchmark workloads may have little or no relevance to the workload current systems actually run, and second, the results of these holistic benchmarks can be heavily influenced by obscure and seemingly tangential factors.

For our analysis, we have created a pair of simple micro benchmarks to analyze read performance. A micro benchmark is appropriate for our research because our goal is to isolate and examine the effects of changes to the read-ahead heuristics on read performance. The

benchmark used in most of the discussions of this paper is defined in Section 4.2. The second benchmark is defined in Section 7. Neither of these benchmarks resembles a general workload – their only purpose is to measure the raw read performance of large files.

We do not attempt to age the file system at all before we run our benchmarks. Aging a file system has been shown to make benchmark results more realistic [24]. For most benchmarks, fresh file systems represent the best possible case. For our enhancements, however, fresh file systems are one of the worst cases. We are attempting to measure the impact of various read-ahead heuristics, and we believe that read-ahead heuristics increase in importance as file systems age. Therefore, any benefit we see for a fresh file system should be even more pronounced on an aged file system.

## 4 The Testbed

In this section, we describe our testbed, including the hardware, our benchmark, and our method for defeating the effects of caching.

### 4.1 The Hardware

The server used for all of the benchmarks described in this paper is a Pentium III system running at 1 GHz, with 256 MB of RAM. The system disk is an IBM DDYS-T36950N S96H SCSI-3 hard drive controlled by an Adaptec 29160 Ultra160 SCSI adapter. The benchmarks are run on two separate disks: a second IBM DDYS-T3690N S96H drive (attached to the Adaptec card), and a Western Digital WD200BB-75CAA0 drive (attached to a VIA 82C686 ATA66 controller on the motherboard). The server has two network interfaces: an Intel PRO/1000 XT Server card running at 1 Gb/s and an 3Com 3c905B-TX Fast Etherlink XL card running at 100 Mb/s.

The clients are Pentium III systems running at 1 GHz, with 1 Gigabyte of RAM, and two network interfaces: an Intel PRO/1000 XT Server card running at 1 Gb/s, and an Intel Pro 10/100B/100+ Ethernet card, running at 100 Mb/s. The 100Mb/s interfaces are for general use, while the 1Gb/s interfaces are only used by the benchmarks.

The gigabit Ethernet cards are connected via a Net-Gear GSM712 copper gigabit switch. The switch and the Ethernet cards use 802.3x flow control, and the standard Ethernet MTU of 1500 bytes. The raw network bandwidth achievable by the server via TCP over the gigabit network is 49 MB/s. This falls far short of the theoretical maximum, but approaches the DMA speed of the PCI bus of the server motherboard, which we measured at 54

MB/s using the `gm_debug` utility provided by Myrinet to test their network cards. (A Myrinet card was installed in the server long enough to run this benchmark, but was not in the system for the other tests.) Only the benchmark machines are attached to the gigabit switch.

All systems under test run FreeBSD 4.6.2. The FreeBSD kernel was configured to remove support for pre-686 CPUs and support for hardware devices not present in our configuration, but we made no other customizations or optimizations beyond what is explicitly described in later sections of the paper. For all tests, the server runs eight `nfsds` instead of the default four, and the clients run eight `nfsiods` instead of the default four.

### 4.2 The Benchmark

The aspect of system performance that we wish to measure is the sustained bandwidth of concurrent read operations; the sustained bandwidth we can obtain *from disk* via the file system when several processes concurrently read sequentially through large files. In this section we describe the simple benchmark that we have designed for this purpose. Although it is simple, our benchmark illustrates the complexity of tuning even simple behaviors of the system.

### 4.3 Running the Benchmark

Before running the benchmark, we create a testing directory and populate it with a number of files: one 256 MB file, two 128 MB files, four 64 MB files, eight 32 MB files, sixteen 16 MB files, and thirty-two 8 MB files. We fill every block in these files with non-zero data, to prevent the file system from optimizing them as sparse files.

The benchmark loops through several different numbers of concurrent readers:

For each  $n$  (1, 2, 4, 8, 16, 32)

- For each file of size  $256/n$  MB, create a reader process to read that file. This process opens the file, reads through it from start to end, and then closes the file. Start all these processes running concurrently.
- Wait until the last reader process has finished. Record the time taken by each reader. The number of MB read divided by the time required for the last reader to finish gives the effective throughput of the file system.



During the first iteration, a single reading process will be created, which will read through the 256 MB file. In the second iteration, two reading processes will concurrently read through different 128 MB files. In the final iteration, 32 reading processes will concurrently read different 8 MB files.

For all of the timed benchmark results shown in this paper, each point represents the average of at least ten separate runs. Unless otherwise mentioned, the standard deviation for each set of runs is less than 5% of the mean (and typically much less).

#### 4.3.1 Defeating the Cache

We wish to benchmark the speed that the file system can pull data from the disk (either explicitly or via read-ahead). To ensure that we measure this (instead of memory bandwidth), we must make sure that the data we read are not already in the cache.

For our particular setup, every set of files contains 256 MB, so a complete iteration of the benchmark requires reading 1.5 GB. Because our clients have 1 GB of RAM and the server has only 256 MB of RAM and files are not re-read until 1.25 GB of other data has been read, none of the data will survive in the cache long enough to have an effect, at least with our current OS. If our clients and servers engaged in cooperative caching, or used a caching mechanism intelligent enough to recognize the cyclic nature of our benchmarks, we would have to take more care to ensure that none of the data are read from cache. Other techniques for ensuring that the data are flushed from the cache include rebooting each client and server between each iteration of the benchmark, unmounting and remounting the file systems used by the benchmark, and reading large amounts of unrelated data until they fill the cache. We experimented with each of these and found that they made no difference to the benchmark results, so we are confident that caching did not influence our benchmarking.

## 5 Benchmarking Traps

Our initial attempt to measure the effect of changes to the FreeBSD NFS server and to experiment with heuristics designed to address some of the behaviors we observed in our earlier NFS trace study was frustrating. Our algorithms were easy to implement, and the diagnostic instrumentation we added to monitor our algorithms confirmed that they were working as intended. However, the results from our benchmarks were confusing and sometimes contradictory. Different runs on the same hardware could give very different results, and the results on dif-

ferent hardware were often puzzling. We had anticipated that the effect of our changes would be relatively small, but we had not expected it to be overwhelmed by unexpected effects. Therefore, before proceeding with the benchmarks, we decided that a more interesting course of action would be to investigate and expose the causes of the variation in our benchmark and see if we could design our experiments to control for them.

### 5.1 ZCAV Effects

Modern disk drives, with few exceptions, use a technique known as zoned constant angular velocity coding (ZCAV) to increase the total disk capacity and average transfer rate [15]. ZCAV can be thought of as an approximation of constant linear density coding, modified to allow the disk to spin at a constant rate and provide an integral number of disk sectors per track. The innermost cylinders of the disk drive contain fewer sectors than the outermost tracks (typically by a factor of 2:3, but for some drives as much as 1:2). The transfer rate between the disk and its buffer varies proportionally; in the time it takes to perform a single revolution, the amount of data read or written to disk can vary by a factor of almost 2 depending on whether the head is positioned at the innermost or outermost track. If the transfer rate between the disk drive and the host memory is greater or equal to the internal transfer rate of the disk at the innermost track, then the effect of the differences in transfer rate will be visible to the host. For contemporary SCSI and IDE drives and controllers, which have host interface bandwidth exceeding their maximum read/write speeds, this difference is entirely exposed. If the proportion between the number of sectors in the innermost and outermost cylinders is 2:3, then reading a large file from the outermost cylinders will take only two-thirds of the time (and two-thirds of the number of seeks, since each track at the outside of the disk contains more sectors).

This effect has been measured and analyzed and is the basis of some well-studied techniques in file system layout tuning [15, 28]. Beyond papers that explicitly discuss methods for measuring and exploiting disk properties, however, mention of this effect is rare.

The ZCAV effect can skew benchmark results enormously, depending on the number of files created and accessed during the benchmark. If two independent runs of the same benchmark use two different sets of files, then the physical location of these files on disk can have a substantial impact on the benchmark. If the effect that the benchmark is attempting to measure is subtle, then it may be completely overwhelmed by the ZCAV effect. It may require a daunting number of runs to statistically separate the effect being measured from the noise intro-



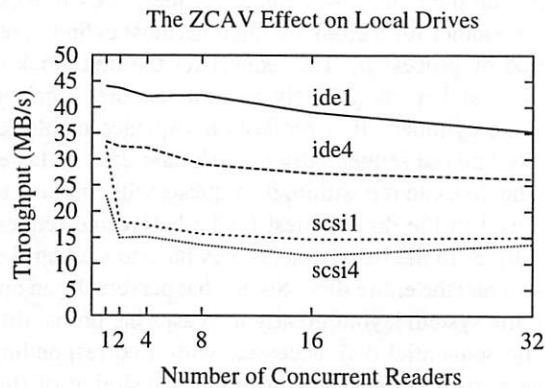


Figure 1: The ZCAV effect. The `scsi1` and `ide1` partitions use the outermost cylinders of the disk, while the `scsi4` and `ide4` use the innermost. As a result, the transfer rates for the `scsi1` and `ide1` partitions are higher than `scsi4` and `ide4`.

duced by using blocks from different areas of the disk.

The ZCAV effect is illustrated in Figure 1, which shows the results of running the same benchmark on different areas of a disk. For our benchmarks, we have divided each of our test disks into four partitions of approximately equal size, numbered 1 through 4. The files in tests `scsi1` and `ide1` are positioned in the outer cylinders of the SCSI and IDE drives, respectively, while the `scsi4` and `ide4` test files are placed in the inner cylinders. For both disks, it is clear that ZCAV has a strong effect. The effect is clearly pronounced for the IDE drive. The SCSI drive shows a weaker effect – but as we will see in Section 5.2, this is because there is another effect that obscures the ZCAV effect for simple benchmarks on our SCSI disk. For both drives the ZCAV effect is more than enough to obscure the impact of any small change to the performance of the file system.

The best method to control ZCAV effects is to use the largest disk available and run your benchmark in the smallest possible partition (preferably the outermost partition). This will minimize the ZCAV effect by minimizing the difference in capacity and transfer rate between the longest and shortest tracks used in your benchmark.

## 5.2 Tagged Command Queues

One of the features touted by SCSI advocates is the availability of *tagged command queues* (also known as *tagged queues*). This feature permits the host to send several disk operation requests to the disk and let the disk execute them asynchronously and in whatever order it deems appropriate. Modern SCSI disks typically

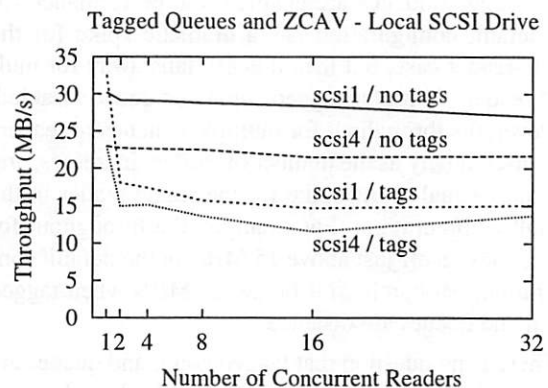


Figure 2: The effect of using tagged queues on SCSI performance. On our test system, disabling tagged queues improves transfer rates substantially for concurrent processes performing long sequential reads.

have an internal command queue with as many as 256 entries. Some recent IDE drives support a feature conceptually identical to tagged command queues, but our IDE drive does not.

The fact that the tagged command queue allows the disk to reorder requests is both a boon for ordinary users and a source of headaches for system researchers. With tagged queues enabled, the disk may service its requests in a different order than they arrive at the disk, and its heuristics for reordering requests may be different from what the system researcher desires (or expects). For example, many disks will reorder or reschedule requests in order to reduce the total power required to position the disk head. Some disks even employ heuristics to reduce the amount of audible noise they generate – many users would prefer to have a quiet computer than one that utilizes the full positioning speed of the disk. The same model of disk drive may exhibit different performance characteristics depending on the firmware version, and on whether it is intended for a desktop or a server.

The SCSI disk in our system supports tagged queues, and the default FreeBSD kernel detects and uses them. We instrumented the FreeBSD kernel to compare the order that disk requests are sent to the disk to the order in which they are serviced, and found that when tagged queues are disabled, the two orders are the same, but when tagged queues are enabled, the disk does reorder requests. Note that this instrumentation was disabled during our timed benchmarks.

To explore the interaction between the FreeBSD disk scheduler and the disk's scheduler, we ran a benchmark with the tagged queues disabled. The results are shown in Figure 2. For our benchmark, the performance is significantly increased when tagged queues are disabled.

When tagged queues are enabled, the performance for the default configuration has a dramatic spike for the single-reader case, but then quickly falls away for multiple readers. With the tagged command queue disabled, however, the throughput for multiple concurrent readers decreases slowly as the number of readers increases, and is almost equal to the spike for the single reader in the default configuration. For example, the throughput for `scsil` levels off just above 15 MB/s in the default configuration, but barely dips below 27 MB/s when tagged command queues are disabled.

There is no question that tagged command queues are effective in many situations. For our benchmark, however, the kernel disk scheduler makes better use of the disk than the on-disk scheduler. This is undoubtedly due in part to the fact that the geometry the disk advertises to the kernel does, in fact, closely resemble its actual geometry. This is not necessarily the case – for example, it is not the case for many RAID devices or similar systems that use several physical disks or other hardware (perhaps distributed over a network) to implement one logical disk. In a hardware implementation of RAID, an access to a single logical block may require accessing several physical blocks whose addresses are completely hidden from the kernel. In the case of SAN devices or storage devices employing a dynamic or adaptive configuration the situation is even more complex; in such devices the relationship between logical and physical block addresses may be arbitrary, or even change from one moment to the next [29]. For these situations it is better to let the device schedule the requests because it has more knowledge than the kernel.

Even for ordinary single-spindle disks, there is a small amount of re-mapping due to bad block substitution. This almost always constitutes a very small fraction of the total number of disk blocks and it is usually done in such a manner that it has a negligible effect on performance.

### 5.3 Disk Scheduling Algorithms

The FreeBSD disk scheduling algorithm, implemented in the `bufqdisksort` function, is based on a cyclical variant of the SCAN or elevator scan algorithm, as described in the BSD 4.4 documentation [13]. This algorithm can achieve high sustained throughput, and is particularly well suited to the access patterns created by the FFS read-ahead heuristics. Unfortunately, if the CPU can process data faster than the I/O subsystem can deliver it, then this algorithm can create unfair scheduling. In the worst case, imagine that the disk head is positioned at the outermost cylinder, ready to begin a scan inward, and the disk request queue contains two requests: one for

a block on the outermost cylinder, requested by process  $\alpha$ , and another for a block on the innermost cylinder, requested by process  $\beta$ . The request for the first block is satisfied, and  $\alpha$  immediately requests another block in the same cylinder. If  $\alpha$  requests a sequence of blocks that are laid out sequentially on disk, and does so faster than the disk can reposition, its requests will continue to be placed in the disk request queue before the request made by  $\beta$ . In the worst case,  $\beta$  may have to wait until  $\alpha$  has scanned the entire disk. Somewhat perversely, an optimal file system layout greatly increases the probability of long sequential disk accesses, with a corresponding increase in the probability of unfair scheduling of this kind.

This problem can be reduced by the use of tagged command queues, depending on how the on-disk scheduler is implemented. In our test machine, the on-board disk scheduler of the SCSI disks is in effect more fair than the FreeBSD scheduler. In this example, it will process  $\beta$ 's request before much time passes.

The unfairness of the elevator scan algorithm is also somewhat reduced by the natural fragmentation of file systems that occurs over time as files are added, change size, or are deleted. Although FFS does a good job of reducing the impact of fragmentation, this effect is difficult to avoid entirely.

The primary symptom of this problem is a large variation in time required by concurrent readers, and therefore this behavior is easily visible in the variance of the run times of each subprocess in our simple benchmark. Each subprocess starts at the same time, and reads the same amount of data, so intuition suggests that they will all finish at approximately the same time. This intuition is profoundly wrong when the default scheduler is used. Figure 3 illustrates the distribution of the individual process times for runs for the benchmark that runs eight concurrent processes, each reading a different 32 MB file. Note that the cache is flushed after each run. The plot of the time required to complete 1 through 8 processes using the elevator scan scheduler on the `ide1` partition shows that the average time required to complete the first process is 1.04 seconds, while the second finishes in 1.98 seconds, the third in 2.94, and so on until the last job finishes after an average of 5.97 seconds. With tagged queues disabled, a similar distribution holds for the `scsil` partition, ranging from 1.18 through 8.54 seconds, although the plot for `scsil` is not as straight as that for `ide1`. The difference between the time required by the fastest and slowest jobs is almost a factor 6 for `ide1`, and even higher for `scsil`.

N-step CSCAN (N-CSCAN) is a fair variation of the Elevator scheduler that prohibits changes to the schedule for the current scan – in effect, it is always planning

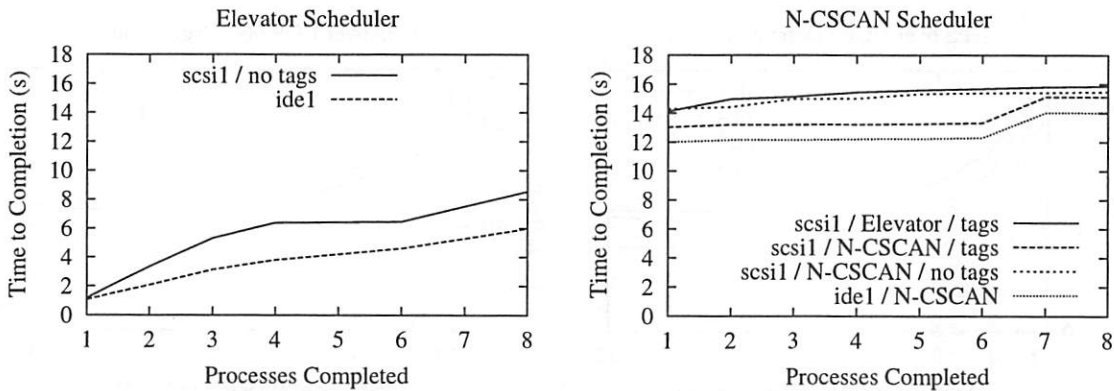


Figure 3: The interaction between tagged queues, the disk scheduling algorithm, and the distribution of average time required to complete a given number of processes. In each run, eight processes are started at the same time, and each reads the same amount of data. Each point on these plots represents the average of 34 runs. Processes run more quickly with the Elevator scan, but the last process takes 6-7 times longer to complete than the first. With the N-CSCAN scheduler, there is less variation in the distribution of running times, but all of the jobs are much slower.

the schedule for the *next* scan [5]. The resulting scheduler is fair in the sense that the expected latency of each disk operation is proportional to the length of the request queue at the time the disk begins its next sweep. Only a small patch is needed to change the current FreeBSD disk scheduler to N-CSCAN. We have implemented this change, along with a switch that can be used to toggle at runtime which disk scheduling algorithm is in use. As illustrated again in Figure 3, this dramatically reduces the variation in the run times for each reader process: for both *ide1* and *scsi1*, the difference in elapsed time between the slowest and the fastest readers is less than 20%.

Unfortunately, fairness comes at a high price: although all of the reading processes make progress at nearly the same rate, the overall average throughput achieved is less than half the bandwidth delivered by the unfair elevator algorithm. In fact, for these two cases, the *slowest* reading process for the elevator scan algorithm requires approximately 50% less time to run than the *fastest* reading process using the N-step CSCAN algorithm. For this particular case, it is hard to argue convincingly in favor of fairness. In the most extreme case, however, it is possible to construct a light workload that causes a process to wait for several minutes for the read of a single block to complete. As a rule of thumb, it is unwise to allow a single read to take longer than it takes for a user to call the help desk to complain that their machine is hung. At some point human factors can make a fair division of file system bandwidth as important as overall throughput.

Also shown in Figure 3 is the impact of these disk scheduling algorithms on *scsi1* when the tagged com-

mand queue is enabled. As described in Section 5.2, the on-disk tagged command queue can override many of the scheduling decisions made by the host disk scheduler. In this measurement, the on-disk scheduling algorithm appears to be fairer than N-step CSCAN (in terms of the difference in elapsed time between the slowest and fastest processes), but even worse in terms of overall throughput.

Although the plots in Figure 3 are relatively flat, they still exhibit an interesting quirk – there is a notable jump between the mean run time of the sixth and seventh processes to finish for N-CSCAN for *ide1* and *scsi1* with tagged queues disabled. We did not investigate this phenomenon.

The tradeoffs between throughput, latency, fairness and other factors in disk scheduling algorithms have been well studied and are still the subject of research [3, 20]. Despite this research, choosing the most appropriate algorithm for a particular workload is a complex decision, and apparently a lost art. We find it disappointing that modern operating systems generally do not acknowledge these tradeoffs by giving their administrators the opportunity to experiment and choose the algorithm most appropriate to their workload.

## 5.4 TCP vs UDP

SUN RPC, upon which the first implementation of NFS was constructed, used UDP for its transport layer, in part because of the simplicity and efficiency of the UDP protocol. Beginning with NFS version 3, however, many vendors began offering TCP-based RPC, including NFS



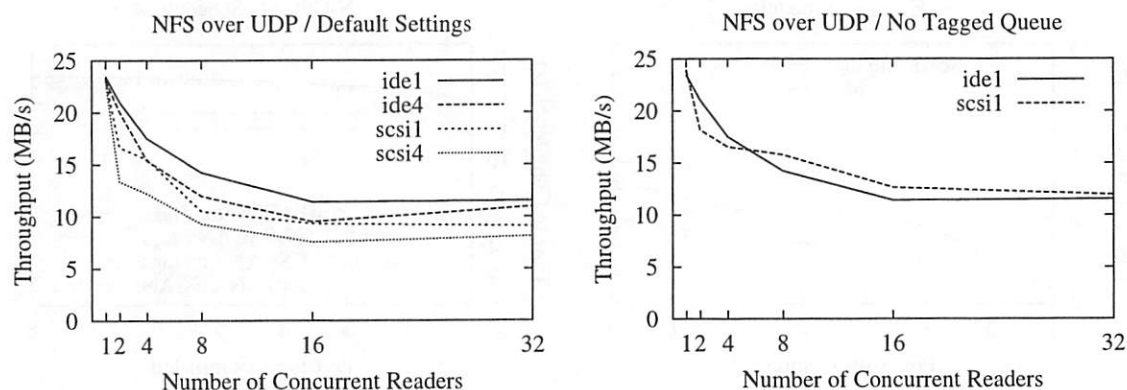


Figure 4: The speed of NFS over UDP, with and without tagged queues. Performance drops quickly as the number of concurrent readers increases. With tagged queues disabled, *scsi1* performance improves relative to *ide1* as the number of concurrent readers increases. Note that the ZCAV effect is still visible.

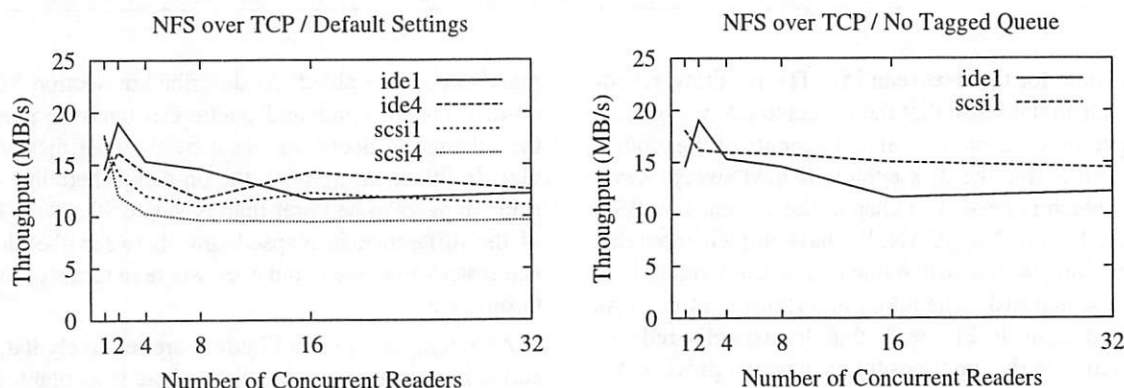


Figure 5: The speed of NFS over TCP, with and without tagged queues. Compared to UDP, the throughput is relatively constant as the number of concurrent readers increases, especially when tagged queues are disabled on the *scsi1*. We do not know why the IDE partitions show a performance spike at two concurrent readers, nor why *ide4* is faster than *ide1* for 16 readers.

over TCP. This has advantages in some environments, particularly WAN systems, due to the different characteristics of TCP versus UDP. UDP is a lightweight and connectionless datagram protocol. A UDP datagram may require several lower-level packets (such as Ethernet frames) to transmit, and the loss of any one of these packets will cause the entire datagram to be lost. In contrast, TCP provides a reliable connection-based mechanism for communication and can, in many cases, detect and deal with packet corruption, loss, or reordering more efficiently than UDP. TCP provides mechanisms for intelligent flow control that are appropriate for WANs.

On a wide-area network, or a local network with frequent packet loss or collision, TCP connections can provide better performance than UDP. Modern LANs are nearly always fully switched, and have very low packet

loss rates, so the worst-case behavior of UDP is rarely observed. However, mixed-speed LANs do experience frequent packet loss at the junctions between fast and slow segments, and in this case the benefits of TCP are also worth considering. In our testbed, we have only a single switch, so we do not observe these effects in our benchmarks.

The RPC transport protocol used by each file system mounted via NFS is chosen when the file system is mounted. The default transport protocol used by `mount_nfs` is UDP. Many system administrators use `amd` instead of `mount_nfs`, however, and uses a different implementation of the mount protocol. On FreeBSD, NetBSD, and many distributions of GNU/Linux, `amd` uses TCP by default, but on other systems, such as OpenBSD, `amd` uses UDP. This choice can



be overridden, but often goes unnoticed.

A comparison of the raw throughput of NFS for large reads over TCP and UDP is given in Figures 4 and 5. Compared to the performance of the local file system, shown in Figure 1, the throughput of NFS is disappointing; for concurrent readers the performance is about half that of the local file system and only a fraction of the potential bandwidth of the gigabit Ethernet. The throughput for small numbers of readers is substantially better for UDP than TCP, but the advantage of UDP is attenuated as the number of concurrent readers increases until it has no advantage over TCP (and in some cases is actually slower). In contrast, the throughput of accesses to the local disk slightly increases as the number of readers increases. We postulate that this is due to a combination of the queuing model used by the disk drive, and the tendency of the OS to perform read-ahead when it perceives that the access pattern is sequential, but also to throttle the read-ahead to a fixed limit. When reading a single file, a fixed amount of buffer space is set aside for read-ahead, and only a fixed number of disk requests are made at a time. As the number of open files increases, the total amount of memory set aside for read-ahead increases proportionally (until another fixed limit is reached) and the number of disk accesses queued up for the disk to process also grows. This allows the disk to be kept busier, and thus the total throughput can increase.

Unlike UDP, the throughput of NFS over TCP roughly parallels the throughput of the local file system, although it is always significantly slower, even over gigabit Ethernet. This leads to the question of why UDP and TCP implementations of NFS have such different performance characteristics as the number of readers increases – and whether it is possible to improve the performance of UDP for multiple readers.

## 5.5 Discussion

As illustrated in Figures 4 and 5, the effects of ZCAV and tagged queues are clearly visible in the NFS benchmarks. Even though network latency and the extra overhead of RPC typically reduces the bandwidth available to NFS to half the local bandwidth, these effects must be considered because they can easily obscure more subtle effects.

In addition to the effects we have uncovered here, there is a new mystery – the anomalous slowness of the `ide1` and `ide4` partitions when accessed via NFS over TCP by one reader. We suspect that this is a symptom of TCP flow control.

## 6 Improving Read-Ahead for NFS

Having now detailed some of the idiosyncrasies that we encountered with our simple benchmark, let us return to the task at hand, and evaluate the benefit of a more flexible sequentiality metric to trigger read-ahead in NFS.

The heuristics employed by NFS and FFS begin to break down when used on UDP-based NFS workloads because many NFS client implementations permit requests to be reordered between the time that they are made by client applications and the time they are delivered to the server. This reordering is due most frequently to queuing issues in the client `nfsiod` daemon, which marshals and controls the communication between the client and the server. This reordering can also occur due to network effects, but in our system the reorderings are attributable to `nfsiod`.

It must be noted that because this problem is due entirely to the implementation of the NFS client, a direct and pragmatic approach would be to fix the client to prevent request reordering. This is contrary to our research agenda, however, which focuses on servers. We are more interested in studying how servers can handle arbitrary and suboptimal client request streams than optimizing clients to generate request streams that are easier for servers to handle.

The fact that NFS requests are reordered means that access patterns that are in fact entirely sequential from the perspective of the client may appear, to the server, to contain some element of randomness. When this happens, the default heuristic causes read-ahead to be disabled (or diminished significantly), causing considerable performance degradation for sequential reads.

The frequency at which request reordering takes place increases as the number of concurrent readers, the number of `nfsiod`s, and the total CPU utilization on the client increases. By using a slow client and a fast server on a congested network, we have been able to create systems that reorder more than 10% of their requests for long periods of time, and during our analysis of traces from production systems we have seen similar percentages during periods of peak traffic. On our benchmark system, however, we were unable to exceed 6% request reordering on UDP and 2% on TCP on our gigabit network with anything less than pathological measures. This was slightly disappointing, because lower probabilities of request reordering translate into less potential for improvement by our algorithm, but we decided to press ahead and see whether our algorithm is useful in our situation and thus might be even more useful to users on less well-mannered networks.

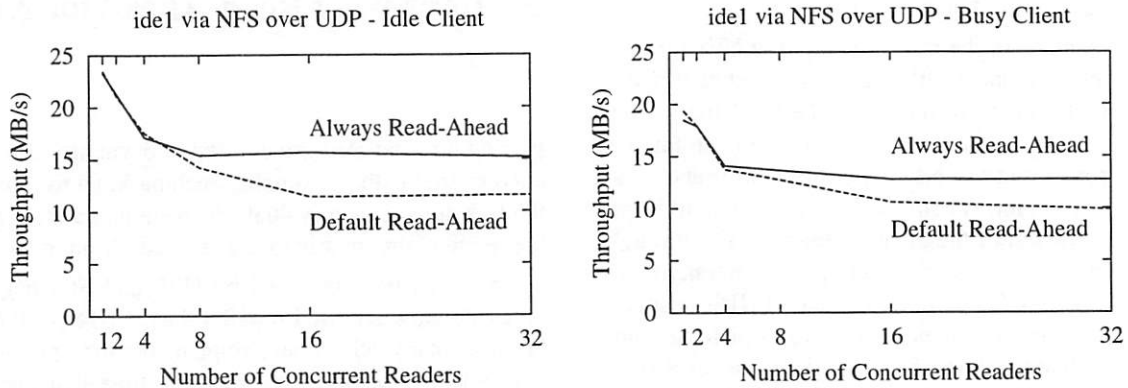


Figure 6: A comparison of the NFS throughput for the default read-ahead heuristic and a hard-wired “always do read-ahead” heuristic. All tests are run using file system `ide1` via UDP. The idle client is running only the benchmark, while the busy client is also running four infinite loop processes.

## 6.1 Estimating the Potential Improvement

Figure 6 shows the NFS throughput for the default implementation compared to the throughput when we hard-wire the sequentiality metric to always force read-ahead to occur. The difference between the “Always Read-ahead” and “Default Read-ahead” lines shows the potential improvement. In theory, for large sequential reads (such as our benchmarks) the NFS server should detect the sequential access pattern and perform read-ahead. As shown in Figure 6, however, for more than four concurrent readers the default and optimal lines diverge. This is due in part to the increased number of packet reorderings that occur when the number of concurrent readers increases, but it is also due to contention for system resources, as we will discuss in Section 6.3.

In our own experiments, we noticed that the frequency of packet reordering increases in tandem with the number of active processes on the client (whether those processes are doing any I/O or not), so Figure 6 also shows throughput when the client is running four “infinite loop” processes during the benchmark. Not surprisingly, the throughput of NFS decreases when there is contention for the client CPU (because NFS does have a significant processing overhead). Counter to our intuition, however, the gap between the “Always Read-ahead” line and the “Default Read-ahead” lines is actually smaller when the CPU is loaded, even though we see more packet reordering.

## 6.2 The *SlowDown* Sequentiality Heuristic

There are many ways that the underlying sequentiality of an access pattern may be measured, such as the metrics developed in our earlier studies of NFS traces. For our

preliminary implementation, however, we wish to find a simple heuristic that does well in the expected case and not very badly in the worst case, and that requires a minimum of bookkeeping and computational overhead.

Our current heuristic is named *SlowDown* and is based on the idea of allowing the sequentiality index to rise in the same manner as the ordinary heuristic, but fall less rapidly. Unlike the default behavior (where a single out-of-order request can drop the sequentiality score to zero), the *SlowDown* heuristic is resilient to “slightly” out-of-order requests. At the same time, however, it does not waste read-ahead on access patterns that do not have a strongly sequential component – if the access pattern is truly random, it will quickly disable read-ahead. The default metric for computing the heuristic, as implemented in FreeBSD 4.x, is essentially the following: when a new file is accessed, it is given an initial sequentiality metric  $seqCount = 1$  (or sometimes a different constant, depending on the context). Whenever the file is accessed, if the current offset  $currOffset$  is the same as the offset after the last operation ( $prevOffset$ ), then increment  $seqCount$ . Otherwise, reset  $seqCount$  to a low value.

The  $seqCount$  is used by the file system to decide how much read-ahead to perform – the higher  $seqCount$  rises, the more aggressive the file system becomes. Note that in both algorithms,  $seqCount$  is never allowed to grow higher than 127, due to the implementation of the lower levels of the operating system.

The *SlowDown* heuristic is nearly identical in concept to the additive-increase/multiplicative-decrease used by TCP/IP to implement congestion control, although its application is very different. The initialization is the same as for the default algorithm, and when  $prevOffset$  matches  $currOffset$ ,  $seqCount$  is incremented as before. When  $prevOffset$  differs from  $currOffset$ , however, the

response of *SlowDown* is different:

- If *currOffset* is within 64k (eight 8k NFS blocks) of *prevOffset* then *seqCount* is unchanged.
- If *currOffset* is more than 64k from *prevOffset*, then divide *seqCount* by 2.

In the first case, we do not know whether the access pattern is becoming random, or whether we are simply seeing jitter in the request order, so we leave *seqCount* alone. In the second case, we want to start to cut back on the amount of read-ahead, and so we reduce *seqCount*, but not all the way to zero. If the non-sequential trend continues, however, repeatedly dividing *seqCount* in half will quickly chop it down to zero.

It is possible to invent access patterns that cause *SlowDown* to erroneously trigger read-ahead of blocks that will never be accessed. To counter this, more intelligence (requiring more state, and more computation) could be added to the algorithm. However, in our trace analysis we did not encounter any access patterns that would trick *SlowDown* to perform excessive read-ahead. The only goal of *SlowDown* is to help cope with small reorderings in the request stream. An analysis of the values of *seqCount* show that *SlowDown* accomplishes this goal.

### 6.3 Improving the *nfsheur* Table

NFS versions 2 and 3 are stateless protocols, and do not contain any primitives analogous to the *open* and *close* system calls of a local file system. Because of the stateless nature of NFS, most NFS server implementations do not maintain a table of the open file descriptors corresponding to the files that are active at any given moment. Instead, servers typically maintain a cache of information about files that have been accessed recently and therefore are believed likely to be accessed again in the near future. In FreeBSD, the information used to compute and update the sequentiality metric for each active file is cached in a small table named *nfsheur*.

Our benchmarks of the *SlowDown* heuristic showed no improvement over the default algorithm, even though instrumentation of the kernel showed that the algorithm was behaving correctly and updated the sequentiality metric properly even when many requests were reordered. We discovered that our efforts to calculate the sequentiality metric correctly were rendered futile because the *nfsheur* table was too small.

The *nfsheur* table is implemented as a hash table, using open hashing with a small and limited number of probes. If the number of probes necessary to find a file handle is larger than this limit, the least recently used file

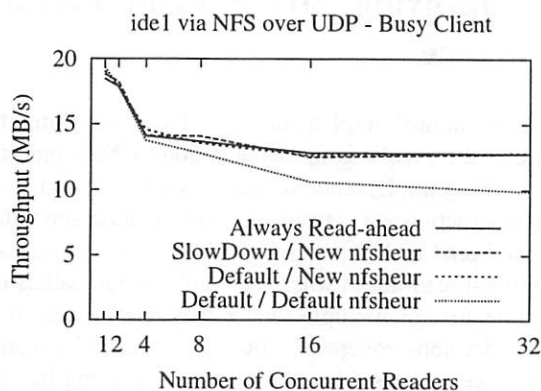


Figure 7: The effect of *SlowDown* and the new *nfsheur* table for sequential reads for disk *ide1* over UDP. The new *nfsheur* increases throughput for more than four concurrent readers, and gives performance identical to the Always Read-ahead case. *SlowDown* makes no further improvement.

handle from among those probed is ejected and the new file handle is added in its place. This means that entries can be ejected from the table even when it is less than full, and in the worst case a small number of active files can thrash *nfsheur*. Even in the best case, if the number of active files exceeds the size of the table, active file handles will constantly be ejected from the table. When a file is ejected from the table, all of the information used to compute its sequentiality metric is lost.

The default hash table scheme works well when a relatively small number of files are accessed concurrently, but for contemporary NFS servers with many concurrently active files the default hash table parameters are simply too small. This is not particularly surprising, because network bandwidth, file system size, and NFS traffic have increased by two orders of magnitude since the parameters of the *nfsheur* hash table were chosen. For our *SlowDown* experiment, it is clear that there is no benefit to properly updating the sequentiality score for a file if the sequentiality score for that file is immediately ejected from the cache.

To address this problem, we enlarged the *nfsheur* table, and improved the hash table parameters to make ejections less likely when the table is not full. As shown in Figure 7, with the new table implementation *SlowDown* matches the “Always Read-ahead” heuristic. We were also surprised to discover that with the new table, the default heuristic *also* performs as well as “Always Read-Ahead”. It is apparently more important to have an entry in *nfsheur* for each active file than it is for those entries to be completely accurate.



## 7 Improving Stride Read Performance

The conventional implementation of the sequentiality metric in the FreeBSD implementation of NFS (and in fact, in many implementations of FFS) uses a single descriptor structure to encapsulate all information about the observed read access patterns of a file. This can cause suboptimal read-ahead when there are several readers of the same file, or a single reader that reads a file in a regular but non-sequential pattern. For example, imagine a process that strides through a file, reading blocks 0,  $x$ , 1,  $x+1$ , 2,  $x+2$ , 3,  $x+3$  ... This pattern is the composition of two completely sequential read access patterns (0, 1, 2, 3, ... and  $x$ ,  $x+1$ ,  $x+2$ ,  $x+3$ , ...), each of which can benefit from read-ahead. Unfortunately, neither the default sequentiality metric nor *SlowDown* recognizes this pattern, and this access pattern will be treated as non-sequential, with no read-ahead. Variations on the stride pattern are common in engineering and out-of-core workloads, and optimizing them has been the subject of considerable research, although it is usually attacked at the application level or as a virtual memory issue [2, 17].

In the ordinary implementation, the *nfsheur* contains a single offset and sequentiality count for each file handle. In order to handle stride read patterns, we add the concept of *cursors* to the *nfsheur*. Each active file handle may have several cursors, and each cursor contains its own offset and sequentiality count. When a read occurs, the sequentiality metric searches the *nfsheur* for a matching cursor (using the same approximate match as *SlowDown* to match offsets). If it finds a matching cursor, the cursor is updated and its sequentiality count is used to compute the effective *seqCount* for the rest of the operation, using the *SlowDown* heuristic. If there is no cursor matching a given read, then a new cursor is allocated and added to the *nfsheur*. There is a limit to the number of active cursors per file, and when this limit is exceeded the least recently used cursor for that file is recycled.

If the access pattern is truly random, then many cursors are created, but their sequentiality counts do not grow and no extra read-ahead is performed. In the worst case, a carefully crafted access pattern can trick the algorithm into maximizing the sequentiality count for a particular cursor just before that cursor dies (and therefore potentially performing read-ahead for many blocks that are never requested), but the cost of reading the extraneous blocks can be amortized over the increased efficiency of reading the blocks that *were* requested (and caused the sequentiality count to increase in the first place).

The performance of this method for a small set of

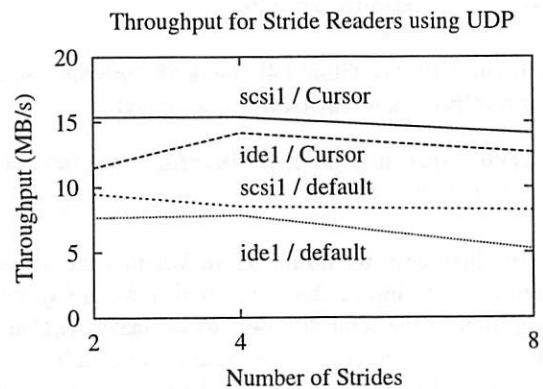


Figure 8: Throughput using the default NFS read-ahead compared to the cursor read-ahead, for reading a 256 MB file using 2, 4, and 8-stride patterns. *scsi1* runs 60-70cursors are enabled. *ide1* is only 50% faster for the 2-stride test with cursors enabled, but 140% faster for the 8-stride test.

stride read patterns is shown in Figure 8 and Table 1. For the “2” stride for a file of length  $n$ , there are two sequential subcomponents, beginning at offsets 0 and  $n/2$ , for the “4” stride there are four beginning at offsets 0,  $n/4$ ,  $n/2$ , and  $3n/4$ , and for the “8” stride there are eight sequential subcomponents beginning at offsets 0,  $n/8$ ,  $n/4$ ,  $3n/8$ ,  $n/2$ ,  $5n/8$ ,  $3n/4$ , and  $7n/8$ . To the ordinary sequentiality or *SlowDown* heuristics, these appear to be completely random access patterns, but our cursor-based algorithm detects them and induces the proper amount of read-ahead for each cursor. As shown in this figure, the time required to read the test files using the cursor-based method is at least 50% faster than using the default method. In the most extreme case, the cursor-based method is 140% faster for the 8-stride reader on *ide1*.

## 8 Future Work

We plan to investigate the effect *SlowDown* and the cursor-based read-ahead heuristics on a more complex and realistic workload (for example, adding a large number of metadata and write requests to the workload).

In our implementation of *nfsheur* cursors, no file handle may have more than a small and constant number of cursors open at any given moment. Access patterns such as those generated by Grid or MPI-like cluster workloads can benefit from an arbitrary number of cursors, and therefore would not fully benefit from our implementation.

In our simplistic architecture, it is inefficient to increase the number of cursors, because every file handle



File System		s = 2	s = 4	s = 8
idel	UDP/Default	7.66 (0.02)	7.83 (0.02)	5.26 (0.02)
	UDP/Cursor	11.49 (0.29)	14.15 (0.14)	12.66 (0.43)
scsil	UDP/Default	9.49 (0.03)	8.52 (0.04)	8.21 (0.03)
	UDP/Cursor	15.39 (0.20)	15.38 (0.15)	14.12 (0.46)

Table 1: Mean throughput (in MB/s) of ten reads of a single 256 MB file using a stride read, comparing the default read-ahead heuristic to the cursor-based heuristic. The cache is flushed before each run. The numbers in parenthesis give the standard deviation for each sample.

will reserve space for this number of cursors (whether they are ever used or not). It would be better to share a common pool of cursors among all file handles.

It would be interesting to see if the cursor heuristics are beneficial to file-based database systems such as MySQL [7] or Berkeley DB [25].

## 9 Conclusions

We have shown the effect of two new algorithms for computing the sequentiality count used by the read-ahead heuristic in the FreeBSD NFS server.

We have shown that improving the read-ahead heuristic by itself does not improve performance very much unless the *nfsheur* table is also made larger, and making *nfsheur* larger by itself is enough to achieve optimal performance for our benchmark. In addition, our changes to *nfsheur* are very minor and add no complexity to the NFS server.

We have also shown that a cursor-based algorithm for computing the read-ahead metric can dramatically improve the performance of stride-pattern readers.

Perhaps more importantly, we have discussed several important causes of variance or hidden effects in file system benchmarks, including the ZCAV effect, the interaction between tagged command queues and the disk scheduling algorithm, and the effect of using TCP vs UDP, and demonstrated the effects they may have on benchmarks.

### 9.1 Benchmarking Lessons

- *Do not overlook ZCAV effects.* To reduce the interference of ZCAV effects on your benchmarks, confine your benchmarks to a small section of the disk, and use the largest possible disk in order to minimize the difference in transfer speed between the innermost and outermost tracks. If the goal of your benchmark is to minimize the impact of seeks or rotational latency, use the innermost tracks. For the best possible performance, use the outermost

tracks.

- *Know your hardware.* Typical desktop workstations use PCI implementations that have a peak transfer speed slower than typical disk drives, and cannot drive a gigabit Ethernet card at full speed.
- *Check for Unexpected Variation.* The disk scheduler can order I/O in a different order than you expect – and tagged command queues can reorder them yet again. This can cause unexpected effects; watch for them.
- *Know your protocols.* Are you using TCP or UDP? Does it make a difference for your test? Would it make a difference in other situations?

## Acknowledgments

The paper benefited enormously from the thoughtful comments from our reviewers and Chuck Lever, our paper shepherd. This work was funded in part by IBM.

## Obtaining Our Software

The source code and documentation for the changes to the NFS server and disk scheduler described in this paper, relative to FreeBSD 4.6 (or later), are available at <http://www.eecs.harvard.edu/~ellard/NFS>.

## References

- [1] Timothy Bray. The Bonnie Disk Benchmark, 1990. <http://www.textuality.com/bonnie/>.
- [2] Angela Demke Brown and Todd C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *The Fourth Symposium on Operating Design and Implementation OSDI*, October 2000.
- [3] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk Scheduling with Quality of Service Guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, Florence, Italy, June 1999.

- [4] Michael Dahlin, Randolph Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, Monterey, CA, 1994.
- [5] Harvey M. Deitel. *Operating Systems, 2nd Edition*. Addison-Wesley Publishing Company, 1990.
- [6] Rohit Dube, Cynthia D. Rais, and Satish K. Tripathi. Improving NFS Performance Over Wireless Links. *IEEE Transactions on Computers*, 46(3):290–298, 1997.
- [7] Paul DuBois. *MySQL, 2nd Edition*. New Riders Publishing, 2003.
- [8] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, pages 203–216, San Francisco, CA, March 2003.
- [9] J. Howard, M. Kazar, S. Menees, S. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [10] Rick Macklem. Not Quite NFS, Soft Cache Consistency for NFS. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 261–278, San Francisco, CA, USA, 17–21 1994.
- [11] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and Performance of the Direct Access File System. In *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA*, pages 1–14, June 2002.
- [12] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *Computer Systems*, 2(3):181–197, 1984.
- [13] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing Company, 1996.
- [14] Larry W. McVoy and Carl Staelin. Imbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.
- [15] Rodney Van Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the Usenix Technical Conference*, January 1997.
- [16] J. Mogul. Brittle Metrics in Operating Systems Research. In *The Seventh Workshop on Hot Topics in Operating Systems: [HotOS-VII]: 29–30 March 1999, Rio Rico, Arizona*, pages 90–95, 1999.
- [17] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-Of-Core Applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, 1996.
- [18] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.
- [19] Thomas M. Ruwart. File System Performance Benchmarks, Then, Now, and Tomorrow. In *18th IEEE Symposium on Mass Storage Systems*, San Diego, CA, April 2001.
- [20] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, Berkeley, CA, 1990.
- [21] Margo Seltzer, Greg Ganger, M. Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *USENIX Annual Technical Conference*, pages 18–23, June 2000.
- [22] Margo I. Seltzer, David Krinsky, Keith A. Smith, and Xiaolan Zhang. The Case for Application-Specific Benchmarking. In *Workshop on Hot Topics in Operating Systems*, pages 102–107, 1999.
- [23] Elizabeth Shriver, Christopher Small, and Keith A. Smith. Why Does File System Prefetching Work? In *USENIX Annual Technical Conference*, pages 71–84, June 1999.
- [24] Keith A. Smith and Margo I. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. In *Proceedings of SIGMETRICS 1997: Measurement and Modeling of Computer Systems*, pages 203–213, Seattle, WA, June 1997.
- [25] Sleepycat Software. *Berkeley DB*. New Riders Publishing, 2001.
- [26] SPEC SFS (System File Server) Benchmark, 1997. <http://www.spec.org/osg/sfs97r1/>.
- [27] Diane Tang. Benchmarking Filesystems. Technical Report TR-19-95, Harvard University, Cambridge, MA, USA, October 1995.
- [28] Peter Triantafillou, Stavros Christodoulakis, and Costas Georgiadis. A Comprehensive Analytical Performance Model for Disk Devices Under Random Workloads. *Knowledge and Data Engineering*, 14(1):140–155, 2002.
- [29] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 90–106. IEEE Computer Society Press and Wiley, 2001.

# Flexibility in ROM: A Stackable Open Source BIOS

Adam Agnew, Adam Sulmicki, \*Ronald Minnich, William Arbaugh

*Department of Computer Science*

*University of Maryland at College Park*

*Advanced Computing Lab, Los Alamos National Lab\**

*Los Alamos, New Mexico*

LANL LA-UR-03-2137

agnew@cs.umd.edu, adam@cfar.umd.edu, rminnich@acl.lanl.gov, waa@cs.umd.edu

## Abstract

One of the last vestiges of closed source proprietary software in current PCs is the PC BIOS. The BIOS, most always written in assembler, operates mostly in 16 bit mode, and provides services that few modern 32 bit operating systems require. Recognizing this, the LinuxBIOS founders began an effort to place a Linux kernel in the ROM of current motherboards— completely removing the legacy BIOS. While the LinuxBIOS effort fully supports Linux, other modern operating systems, e.g. \*BSD, and Windows 2000/XP, could not be directly supported because of their reliance on a few services provided by those legacy BIOSes. In this paper, we describe how we have combined elements of the LinuxBIOS, the Bochs PC emulator, and additional software to create the first open source firmware for the IBM PC capable of booting most modern operating systems.

## 1 Introduction

The personal computer (PC) basic input output system (BIOS) remains one of the last bastions of closed source software. The reasons for this are many, but the most prominent is that many of the hardware interfaces in modern PC chipsets are covered by non-disclosure agreements. NDAs greatly limit the set of systems that an open source BIOS could support. Additionally, technical barriers to entry at the firmware/BIOS level are significant. For example, debugging is considerably more difficult, and the tools to permit control over the processor at the firmware level, e.g. an in-circuit emulator, are very expensive.

These technical and non-technical barriers, coupled with a near monopoly in the BIOS market, have prevented innovation from occurring in a key element of the personal

computer. As a result, commercial BIOSes continue to be written entirely in assembler, run mostly in 16 bit mode, and provide few services beyond the interrupts initially provided by the BIOS in the 1980's.

The LinuxBIOS effort at Los Alamos National Labs sought to change that situation. Frustrated by the problems created by the BIOS in large sets of computing systems, i.e. clusters, the LinuxBIOS team began to create their own open source BIOS. The LinuxBIOS also sought to place an operating system kernel, e.g. Linux, in the ROM of a PC motherboard.

The reason for putting a full kernel in place of the BIOS is simple: Linux does a far better job of detecting and configuring hardware than standard BIOSes do, and its “footprint” in memory is not really that much larger. As of 1999 there are many systems (kmonite, kexec, bootimg, LOBOS) that allow Linux to boot another operating system. The result is that LinuxBIOS can load a kernel over any device, network, protocol, and file system that Linux supports— compelling reasons to use Linux as a boot program.

Even with the considerable technical barriers, LinuxBIOS now runs on numerous motherboards. While it remains unlikely that LinuxBIOS will ever become available on all PC motherboards, the current success of the project is significant.

LinuxBIOS is actually divided into two parts. The first part is a small open-source system startup code. The second part is the Linux kernel itself. Over time, users have found that they can use the first part without the second, and the Linux kernel has been replaced by many different types of software, from Etherboot to 9load, the Plan 9 loader. This has led to a confusing situation with respect to naming: in the early days of LinuxBIOS, the

software implicitly included the Linux kernel. Nowadays, the name “LinuxBIOS” has gradually come to mean only the small open source system startup code. For the rest of the paper, when we use the term “LinuxBIOS” we mean the small startup code, not the full startup code plus Linux kernel combination.

While the original motivation for LinuxBIOS was focused solely on improving the management of large computing clusters, numerous developers viewed LinuxBIOS as a means to provide additional features in the BIOS such as security, and support for additional operating systems beyond Linux. In this paper, we describe one of those efforts— an open source stackable BIOS.

The elements of the stackable BIOS are shown in Figure 1. The first element is LinuxBIOS. LinuxBIOS performs all of the steps necessary to initialize the hardware on the motherboard. The next step is ADLO, the “Adhesive LOader.” This is stackable BIOS software specifically written to serve as the “glue” between the LinuxBIOS and the next element, the Bochs BIOS. Bochs is a highly portable open source x86 PC emulator which includes emulation of the x86 CPU and common I/O devices. Bochs also includes a custom BIOS which provides the legacy BIOS functionality needed to boot modern operating systems such as Linux, OpenBSD, and Windows 2000 using the GNU Grub bootloader.

The resulting BIOS which comes from the combination of LinuxBIOS, ADLO, and the Bochs BIOS has proven to be quite slim and fast.

LinuxBIOS, ADLO, BochsBIOS, and even the Video BIOS (typically 64KB) can take up less than 175KB in size (Figure 2). Because alignment data pads out many of the images, a size of less than 100KB can be achieved.

In the remainder of this paper, we present the details of how elements of two relatively mature open source projects (LinuxBIOS and Bochs) were combined with additional GPL'd software written by the authors to completely eliminate the need for a proprietary legacy BIOS.

## 2 LinuxBIOS

LinuxBIOS is a GPLed project started at the Cluster Research Lab, a division of the Advanced Computing Laboratory (ACL), in Los Alamos National Labs. A computer cluster is a group of computers connected to work together as a parallel computer. The Cluster Research Lab performs research and development in oper-

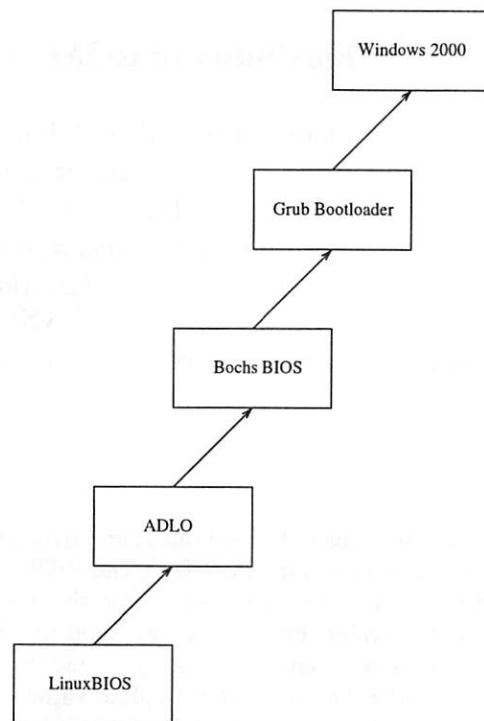


Figure 1: Stackable BIOS Overview

ating systems and cluster design in order to improve the way clusters are built, managed, and used.

LinuxBIOS was created to fill a need in the Computer Research Lab for an open source BIOS. Currently, x86 motherboards ship with BIOSes supplied by vendors such as Phoenix Technologies and American Megatrends. But these BIOSes had shortcomings which made them impractical for cluster use.

The ACL team found that existing BIOSes do a poor job of setting up a PC for modern OSes. They found:

- BIOSes which configure memory in a suboptimal way. For example, some BIOSes were discovered to use memory capable of CAS2 speeds at a much slower CAS3 setting.
- incorrect configuration of PCI address spaces that open up security holes when memory-mapped cards (e.g. Myrinet) are used.
- suboptimal assignment of interrupt requests. Some BIOSes were found to share IRQs between multiple devices when such sharing was not required. This sharing of IRQs added latency upon interrupts



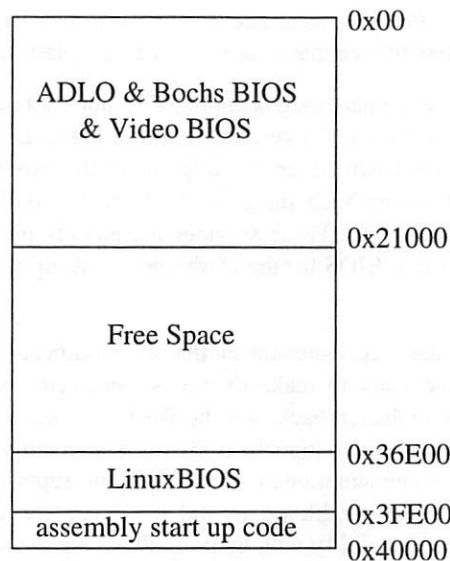


Figure 2: Typical LinuxBIOS + ADLO + BochsBIOS EEPROM map for a 512k part.

needlessly.

- incorrectly configured BIOS tables such as the \$PIR table, so that the critical information Linux needs for IRQ assignment is not available. So pervasive are some of the errors that they have impacted the Linux 2.4.19 kernel; in this version, GEODE IRQ setup was changed, incorrectly, to accommodate some broken motherboards.
- incorrect ID strings in the BIOS, for example the string “Mainboard vendor name here” is in the BIOS instead of the mainboard vendor name
- no way to upgrade the BIOS from a modern OS (some vendors ship DOS diskettes for a BIOS upgrade; others require that you boot a Windows 3.1 CD and run an upgrade program).
- no way to preserve CMOS settings when a BIOS is upgraded (one vendor recommended we record the settings on paper and then re-enter them for each machine; a difficult task on a 960-node system with no keyboard or console).

There are also structural problems with BIOSes that derive from the requirements for supporting DOS. One of these is the requirement that the BIOS zero memory. This requirement makes post-mortem debugging virtually impossible, as a reset results in erasing the memory image of the previously running operating system.

Commercial BIOSes also have problems accommodating custom built hardware, and can take considerable time to boot. Finally, when bugs or errors crop up in a commercial BIOS, it is almost impossible to fix them.

These short-comings made it obvious that an open source BIOS such as LinuxBIOS is needed, especially for the fast paced innovations which can make the next generation of clusters successful.

To accomplish its task, LinuxBIOS is placed in the computer’s EEPROM chip (Electrically Erasable Programmable Read Only Memory) where normally a vendor supplied BIOS would reside. LinuxBIOS completely replaces the vendor BIOS; no fragment of the vendor BIOS is left once LinuxBIOS is installed. The LinuxBIOS binary itself is small, typically only 36 kilobytes in size depending on configuration choices. Then, a Linux kernel is placed in the EEPROM chip to act as a bootloader (Figure 3).

The Linux kernel is used for this task because it supports a wide array of hardware that can be used to acquire the binaries necessary for further booting operating systems. The Linux kernel is well suited for this task because it is, for the most part, written to avoid a need for legacy BIOS services. This lack of dependencies reduces the number of services LinuxBIOS must provide and thus results in a small and compact BIOS.

With the complexity and latency of commercial BIOSes removed, the LinuxBIOS developers discovered they were able to accomplish the entire bootstrap process in a matter of seconds.

Pretty soon, the LinuxBIOS developers found there was demand not only from those building clusters, but also from a computer enthusiast community, and even more notably, motherboard vendors who were hopeful of a future where they wouldn’t have to pay a royalty to BIOS vendors.

This created several problems for the LinuxBIOS developers if they wished to create an open source BIOS for a wider audience.

- Most motherboards ship with 256Kbyte EEPROM parts. Even under extreme compression, the Linux kernel could not fit in such a small flash.
- LinuxBIOS was kept small and fast by not supporting legacy PC BIOS services. The Linux Kernel used with LinuxBIOS had to be slightly modified

to avoid using these legacy services. Unfortunately, many of the other operating systems that motherboard vendors and computer enthusiasts want to run require these services. It is not practical to modify the BIOS interface of each of these operating systems. This is especially true for operating systems which are closed source.

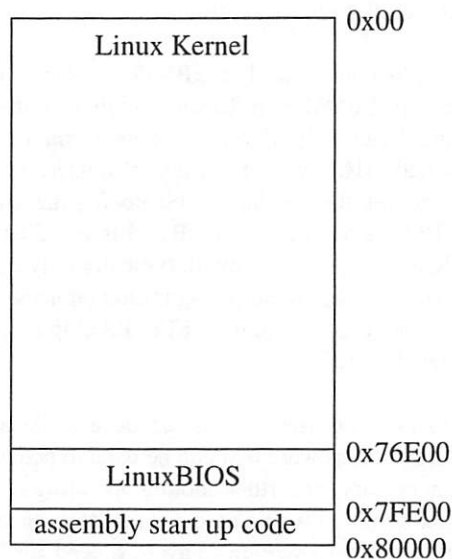


Figure 3: Typical LinuxBIOS EEPROM map (using a Linux Kernel) for a 512k part.

### 3 Bochs

We were able to solve both of these problems by taking advantage of another open source project, Bochs.

Bochs is an LGPLed project, sponsored by Mandrake-Soft, which serves as a highly portable x86 emulator written in C++. Bochs is a true, complete emulator in that it fully emulates an x86 CPU, common I/O devices such as floppy and hard drives, and an x86 BIOS.

The component of Bochs which we found useful in our plans to add PC BIOS services to LinuxBIOS was its BIOS. The Bochs BIOS had many attributes which made it especially appealing for our needs.

- None of the hardware emulated by Bochs is implemented in their BIOS layer. The BIOS was designed to interact with the hardware through true device calls and aimed to conform to up to date standards. This was advantageous to us in that few

modifications were needed for the Bochs BIOS to utilize the real hardware on our target platform.

- Since the hardware is emulated, it does not need to be “turned on” like on a real x86 platform. This means Bochs does not implement the process of initializing such things as CPU cache, IDE controllers, etc. These services are already provided by LinuxBIOS for the motherboards it supports.

Further, these attributes meant that the modifications we found necessary to make to Bochs were prime candidates for inclusion back into the Bochs source. All the changes we needed to make to Bochs were merely more correct implementations of the BIOS interrupts it provided. These modifications did not break the Bochs BIOS’ compatibility with their emulated hardware, and allowed it to work more correctly with the real hardware on our platform as fewer assumptions were made.

This means that further maintenance of the BIOS layer of our solution can stay within the Bochs source and LinuxBIOS can continue to leverage itself effortlessly off of the further hard work of the Bochs PC emulator developers.

## 4 BIOS Dependencies in Modern Operating Systems

The BIOS services required to boot some modern operating systems are plentiful. Most are trivial and hardly worth mentioning, such as probing for keyboard presence or the timer interrupt. But there are four main services which proved critical to booting both operating systems like Windows 2000 and the Linux Kernel 2.4 series. These services comprised the video BIOS functions, hard drive services, memory sizing, and providing a PCI table. Many more services are useful, but they are still new standards which are optional to use in modern operating systems, see Table 1.

The following section outlines the steps taken to make ADLO and Bochs BIOS functionality closer to that of standard commercial BIOSes and distinguish this functionality from that of its base LinuxBIOS, see Table 2.

### 4.1 Video BIOS Functions

Among the first of the issues dealt with was providing Video BIOS functions. In a stock LinuxBIOS setup, the issue is averted by rerouting the console output to a serial

<i>BIOS Feature Needed</i>	<i>Standard 2.4.x Linux Kernel</i>	<i>Modified for LinuxBIOS 2.4.x Linux Kernel</i>	<i>Windows 2000[3]</i>	<i>Windows XP[4]</i>	<i>FreeBSD[5]</i>
Int13 Handling	yes	no	yes	yes	yes
Int15 ax=E820 Map*	yes	no	yes	yes	yes
PnPBIOS	no	no	no	no	no
PCI Table	yes	yes	yes	yes	yes
Video BIOS	no	no	yes	yes	yes
ACPI	no	no	no	no	no
APM	no	no	no	no	no

Table 1: Operating system dependencies on BIOS interrupt functionality.

\* While Int15 function ax=E820 is not itself needed, it is the most modern and popular of the Int15 memory sizing functions of which at least one is needed.

<i>Feature</i>	<i>Standard PC BIOS[2]</i>	<i>LinuxBIOS using Linux Kernel</i>	<i>LinuxBIOS using Etherboot</i>	<i>LinuxBIOS using ADLO and Bochs BIOS</i>
Int13 Handling	yes	no	no	yes
Int15 ax=E820 Map	yes	no	no	yes
PnPBIOS	yes	no	no	no
PCI Table	yes	yes	yes	yes
Video BIOS	yes	yes	yes	yes
ACPI	yes	no	no	no
APM	yes	no	no	no

Table 2: Features provided by a variety of BIOS configurations

console. Regrettably, this luxury was rarely available in other operating systems, so the issue had to be resolved in another way.

In most commercial BIOSes, Video BIOS functionality is provided by an add-in card containing an expansion ROM. This ROM is found while probing the IO buses for expansion ROMS. When a Video BIOS is found, it is copied into memory area 0xC0000 to 0xC7FFF and executed. The Video BIOS in turn claims control of Int10.

On the motherboard we developed with, however, the video chipset was integrated and the Video BIOS was stored with the commercial BIOS on the motherboard's EEPROM. As these images are to be replaced with our code, the Video ROM had to be acquired and loaded in a different fashion.

We accomplished this task by extracting the Video BIOS from memory before installing LinuxBIOS. Since we can expect the Video BIOS to be stored at address 0xC00000, we can boot into Linux using the commercial BIOS shipped with the motherboard and extract this image using 'dd' and the /proc/kcore memory interface. The total size of the image is determined by the 3rd byte, which represents the number of 512 byte blocks. Because the /proc/kcore interface is an ELF image, an off-

set must be taken into account in order to skip the ELF header. This extraction method will work easily on most integrated and non integrated motherboards alike. Once the video BIOS image is extracted it is stored in the EEPROM, see Figure 2.

Once the Video BIOS is in place, both the Linux and Windows operating systems will use it for such functions as preliminary output when booting, VESA compliance probing, and setting video modes.

## 4.2 Hard Drive Services and Interrupt 13

Historically, hard drive services have been the most complex and frustrating of the BIOS services for both the BIOS and operating system developers. As hard drives have grown in capacity, they have consistently been the first of the devices to feel the strain imposed by the instruction width of architectures. As a result, there have been several evolutionary steps in addressing modes and the complexity of the Int13 services has become staggering.

Speed constraints, buggy BIOSes, addressing mode revisions, and an increasing variety of devices has compelled designers of modern operating systems to remove

dependency on Int13 for hard drive functions for the most part. Instead, they communicate with the drive controllers directly, in “polled I/O mode”, and only the bootloaders, which are constrained by space from having drivers for a variety of controllers, still depend on these services heavily.

In this regard, a LinuxBIOS configuration with the Linux Kernel housed in ROM with LinuxBIOS is again at an advantage. The Linux Kernel has all the necessary drivers for properly communicating with a variety of drive controllers. The only issue that needs to be addressed then is waiting for the hard drives to spin up to operational speed. While the boot times of most commercial BIOSes are long enough that it is taken for granted that hard drives will be fully spun up and available before they are needed, the fast boot times of LinuxBIOS makes this assumption incorrect. To correct the mistake, the kernel is patched to assume drives are present before they are available for actual I/O.

As we aimed to boot unmodified operating systems and bootloaders such as LILO, GRUB, and the Windows 2000 Bootloader, we needed to support a much larger set of Int13 functionality.

The Bochs BIOS provided an excellent starting point toward this end; it implemented virtual devices and defined the Int13 functions necessary for bootloaders and operating systems to interface with them. When Bochs virtualizes these devices under a host operating system, the host operating systems already deals with timing issues with the actual physical devices. For this reason, timing with Int13 had not been taken into account directly in the Bochs BIOS. As our BIOS does not include such a virtualization, we implemented proper handling of timing issues to avoid such complications as trying to access devices before they were spun up to operating speed or reading buffers before they could be filled by the hard drive.

### 4.3 Memory Sizing

Like hard drive addressing modes, memory sizing has also gone through evolutionary steps as instruction widths and the cost of memory has dramatically changed. While memory sizing used to be accomplished by simply returning the amount of 1K size blocks available through Int15 function `ah=088h`, limiting the returned amount to 64MB, the service is now commonly performed through subsequent calls to Int15 function `ax=0e820h` in order to fill sections of a table[8].

A typical Int15 `ax=E820` memory table contains several entries, each describing a range in memory with unique characteristics, see Table 3. In this example, the first range identifies the memory from the beginning to 640K as available in keeping with the convention of Lower Memory. Next, a range up to the first megabyte is reserved for BIOS use. The other segments map the rest of available memory, reserving some sections for ACPI use.

An operating system is only interested in the memory ranges which it can use, see table 4. To reduce complexity, we removed sections from the table which were reserved. LinuxBIOS coupled with the Bochs BIOS does not provide ACPI functions, so those sections do not need to be explicitly identified either. The simplified Int15 `ax=E820` table was able to get us past issues in development that dealt with an AMI compatible CMOS which we were unable to provide.

### 4.4 PCI Tables

The original PCI design for interrupts was very simple. PCI supports four interrupts lines, A, B, C, and D. The B, C, and D lines are reserved for multifunction cards, so that only the A line can be used for single-function cards. The problem is that most cards are single-function; most cards drive the A line. Most cards would have to share the A interrupt, leading to high interrupt latency.

This is the reason for one of the more distressing kludges related to PCI busses. Vendors decided to remap the A, B, C, and D lines on the motherboard so as to more evenly distribute the interrupt load. The A line for a given slot may actually be mapped to the B, C, or D interrupt pins on the interrupt controller.

How can an operating system tell, from reading the PCI bus configuration, how the lines are actually mapped? In practice, it can not. Hence the need for the PCI tables.

There are two types of PCI tables: the older \$PIR table, and the newer SMP table. The BIOS must supply both these tables, the older one for older operating systems or newer OSes configured to use the old table (e.g. Linux in uniprocessor mode); and the newer one which is required for correct functioning of an SMP OS.

Each table has basic information about the motherboard, including a variable-length list of PCI slots and the mapping of the four PCI slot interrupt lines to the actual interrupt lines on the interrupt controller. This information can be used to determine, for a given interrupt line,



BIOS-e820:	0000000000000000	-	000000000000A0000	(usable)
BIOS-e820:	000000000000F0000	-	00000000000100000	(reserved)
BIOS-e820:	00000000000100000	-	000000001FFF0000	(usable)
BIOS-e820:	000000001FFF0000	-	000000001FFF3000	(ACPI NVS)
BIOS-e820:	000000001FFF3000	-	0000000020000000	(ACPI data)
BIOS-e820:	00000000FFFF0000	-	0000000100000000	(reserved)

Table 3: Int15 ax=E820 memory map in a commercial BIOS

BIOS-e820:	0000000000000000	-	000000000000A0000	(usable)
BIOS-e820:	00000000000100000	-	000000001FFF0000	(usable)

Table 4: Int15 ax=E820 memory map in LinuxBIOS using Bochs BIOS

which card or cards is the source of the interrupt. The operating system has to build an internal mapping of the card IRQs so that it can make this determination.

One of the more troublesome aspects of the tables is that they contain errors. For example, some Geode motherboards have the low order bit set incorrectly. This led to an incorrect patch being applied to the Linux 2.4.19 release kernel which broke the kernel on correct motherboards so the kernel would work on incorrect motherboards. As of this writing this mistake has not been corrected.

#### 4.5 Future Work in ADLO and Bochs BIOS

In recent years, new standards have emerged and been quickly adopted by both BIOS manufactures and the developers of operating systems.

The Advanced Configuration and Power Interface (ACPI) is a particularly powerful specification that aims to take over the services performed by PnPBIOSes, multiprocessor tables, and Advanced Power Management (APM). However, support for ACPI has just recently been started in the Linux Kernel 2.5 series. Many Windows varieties already extensively make use of ACPI when available, but it is still not necessary for booting these operating systems. Int15 function ax=E820 is the first of the services from the ACPI specification which we have chosen to implement and a full implementation of ACPI would be a worthy goal in further development of ADLO and Bochs BIOS.

As ACPI aims to replace the functionality of APM, PnPBIOS, and multiprocessor tables, adding support in ADLO and Bochs BIOS to support these functions would equate to adding a layer of services which are not necessary now and will become legacy in the near future.

Other directions ADLO and Bochs BIOS development will likely take is full support for multiprocessor systems (this has been added into the Bochs BIOS for use in Bochs, but remains to be tested on real hardware), and a virtual Video BIOS to allow for serial consoles like on many server class motherboards.

## 5 The Bootstrap Process

Though LinuxBIOS and the Bochs BIOS are an open source take on the PC BIOS, neither intended to be a drop in replacement for commercial BIOSes. By stacking and executing the LinuxBIOS, ADLO, and Bochs BIOS components in order, the foundation is in place for an open source BIOS to finally be a viable alternative to commercial BIOSes on many x86 PCs. A description of the tasks performed by each component in this advantageous configuration follows.

### 5.1 LinuxBIOS

The guiding philosophy of LinuxBIOS is simple: "Let the Linux kernel do it." In other words, if Linux can perform some task such as device initialization, there is no need for any other software to do that work. We have found that in most cases Linux will redo work that the BIOS did anyway; or, still worse, Linux has to redo work that the BIOS did since in so many cases the BIOS gets it wrong. The BIOS should only do the bare minimum of work that Linux can't do. By doing the bare minimum, LinuxBIOS becomes fast and small.

In the procedure of booting up the computer to a usable state, LinuxBIOS performs many tasks. First among these tasks is to set up and initialize memory; and set up the serial consoles so debugging information is available. The resources which the LinuxBIOS developers

have available to them on the chipsets for which they develop are often vague, scant, or simply non-existent. With this in mind, it is fitting for a serial console to be such a high priority.

Once the DRAM is initialized, LinuxBIOS can continue in C code. This is a welcome feature when faced with thousands of lines of x86 assembly.

From this point, LinuxBIOS can set up Memory Type Range Registers (MTRRs) on the CPU(s), making the cache of the CPU(s) available and thus speeding up execution considerably. LinuxBIOS is also responsible for creating an IRQ routing table, and initializing individual hardware components on the motherboard. These often include an IDE controller, keyboard, southbridge, etc. Again, only the bare minimum initialization is done; Linux does the rest.

When LinuxBIOS is finished initializing hardware, it then looks in the contents of the ROM to find a next stage in the boot process. With the traditional approach of LinuxBIOS, this would be a Linux Kernel. This configuration is what provides such phenomenal records such as 3 seconds from power-on to a bash prompt.

There are however other extensions to LinuxBIOS which make full use of its modular design. A popular choice among these would be to load Etherboot, which gives the bootstrap process the ability to retrieve the next stage in the bootstrap process (either another component to further enhance the boot strap process, or the final operating system itself)[7].

## 5.2 ADLO

Indeed, instead of executing a Linux Kernel at this stage, we want to load the Bochs BIOS to provide standard commercial PC BIOS interrupt support.

In order to do this effectively, we built a small wrapper program around the Bochs BIOS to transfer valuable information from LinuxBIOS to the Bochs BIOS without having to make modification to the Bochs BIOS. We have named this small wrapper ADLO, the "ADhesive LOader." ADLO allows us to leverage the capabilities of the Bochs BIOS without making modifications to it.

ADLO is responsible for making sure the ROMs that make up the Bochs BIOS and the VGA BIOS are stored at the expected addresses. It also performs the task of copying the Bochs BIOS from its original location into "Shadow RAM." In addition, LinuxBIOS stores some

tables (such as a memory map and the IRQ routing table) in a format designed to be practical across architectures, but not conforming to the format in which they're normally stored on a commercial PC BIOS. ADLO converts these tables back to a format easily understood by the Bochs BIOS as it is implemented presently.

In the same vain, the Bochs BIOS was written for the Bochs emulation environment, and was written to emulate an AMI BIOS. To accomplish this goal, configuration of the Bochs BIOS is done by storing and retrieving configuration data in the CMOS as a real BIOS would do. ADLO is responsible for storing some of this data in the CMOS before executing the Bochs BIOS for parameters such as primary boot device and floppy size.

## 5.3 Bochs BIOS

The primary job of the Bochs BIOS is to set up the Interrupt Vector Table, and supply an entry point for each of its BIOS services. The interrupt vector table is stored from memory location 0000:0000h up to 0000:03FCh and contains a maximum of 256 vectors, each 4 bytes wide.

For instance, if we wanted to save the interrupt vector for our newly implemented Interrupt 13, the vector would be saved as the 19th entry ( $13h = 19$ ) in the interrupt vector table. Since each interrupt vector is 4 bytes wide, the vector for Int13 will be stored at address 0x4C ( $19 * 4$ ).

The interrupt handler for Interrupt 13 is placed at offset 0xE3FE within the BIOS image. The BIOS image is placed at segment 0xF000 in memory. Therefore, the four bytes at this offset can merely contain the segment (0xF000) as the high four bytes and the offset (0xE3FE) as the low four bytes.

## 5.4 Bootloader and Operating System

After the Bochs BIOS has established its interrupt vector table, we rely on a good foundation of BIOS interrupt services to insure that everything continues to run smoothly. To date, we have managed to refine the BIOS services until booting of Linux kernels, which have not been modified for LinuxBIOS compatibility, and Windows 2000 have been possible. Once these services had been refined that far, OpenBSD began to boot through to completion as an added reward. Other modern operating systems, for instance FreeBSD and Windows XP, will require further work while still more have gone so far unexplored. All bootloaders we have tried work flaw-

lessly, including the popular Grub and Lilo.

## 5.5 Hardware Support

To date, the boot strap process outlined in this paper has only been tested on motherboards featuring the SIS630 North/South Bridge chipset. This chipset was chosen as a stable basis for our development efforts due to the long history of support SiS Semiconductors has shown the LinuxBIOS team.

Using any other mainboards in replace of our test system would comprise four steps.

- Insuring that the mainboard is already supported by the LinuxBIOS project[6], or adding support to LinuxBIOS for that mainboard.
- Insuring that LinuxBIOS takes the extra step of initializing the IDE controller. As the Linux Kernel is in most cases capable of this task, it has not already been done for all motherboards with LinuxBIOS supports.
- Extracting of the IRQ routing table for use by ADLO to convey a correct table to the Bochs BIOS.
- Extraction the VGA ROM for use by ADLO if a graphical console is desired.

There are several other hardware issues of interest which are not yet addressed by our efforts.

The first of these is USB support. There are several projects which would benefit greatly from a small open source USB stack. As PS/2 ports have been considered legacy devices for several years now and motherboards are now being sold to consumers which do not feature PS/2 ports at all, the need for simple USB Human Interface Device (HID) support as well as generic USB Storage support has become apparent.

SMP support has not yet been explored. However, LinuxBIOS itself supports SMP on many motherboards, and SMP support has been integrated into Bochs and Bochs BIOS. We expect the integration of the two to be trivial.

## 6 Conclusions

While open source operating systems on the PC have flourished, the same can not be said for the firmware or

BIOS. The reasons for this are many, but the lack of an open source and general purpose BIOS has limited innovation in an important space of the personal computer.

The open source stackable BIOS, described in this paper, eliminates the proprietary BIOS from numerous motherboards, and as a result, a number of new projects can be started within the BIOS space. For example, low level cryptographic support can now be easily added for strong pre-boot authentication, secure remote console support, and secure configuration management are just some of the possible new efforts that can be started now that a general purpose open source BIOS exists.

## Source Availability

Instructions for obtaining the source code for this project is located at <http://www.missl.cs.umd.edu/Projects/sebos/main.shtml>

## Acknowledgments

Research at the University of Maryland on this project was funded in part by the Composable High Assurance Trusted Systems Program at DARPA via AFRL/IF-NY contract F30602-01-2-0535

Research conducted in the Cluster Research Lab at the Los Alamos National Labs was funded in part by the Mathematical Information and Computer Sciences (MICS) Program of the DOE Office of Science and the Los Alamos Computer Science Institute (ASCI Institutes). Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. Los Alamos, NM 87545 LANL LA-UR-03-2137.

We would like to thank the following people for their hard work, suggestions, and encouragement.

The bochs developers have been an invaluable and helpful group, especially Christopher Bothamy.

We would also like to extend a special thank you to the LinuxBIOS developers, of whom Eric Biederman and Ollie Lho were especially helpful.

The guidance and suggestions of Pascal Dornier and David Sankel made sure that we overcame hurdles

swiftly and easily in bringing all of this work together.

## References

- [1] Ron Minnich, James Hendricks, Dale Webster. The Linux BIOS. The Fourth Annual Linux Showcase and Conference, October 2000.  
<http://www.acl.lanl.gov/linuxbios/papers/als00/linuxbios.pdf>
- [2] Phoenix Technologies Ltd. The PhoenixBIOS 4.0 Revision 6 User's Manual.  
<http://www.phoenix.com/resources/userman.pdf>
- [3] Adam Sulmicki. A Study of BIOS Interrupts as used by Microsoft Windows 2000.  
<http://www.missl.cs.umd.edu/Projects/sebos/winint/index1.html>
- [4] Adam Sulmicki. A Study of BIOS Interrupts as used by Microsoft Windows XP.  
<http://www.missl.cs.umd.edu/Projects/sebos/winint/index2.html>
- [5] Bruce M. Simpson. FreeBSD BIOS Coupling on i386.  
<http://www.incunabulum.com/code/projects/freebsd/freebsd-bios-interaction.txt>
- [6] The LinuxBios Status Guide.  
<http://www.linuxbios.org/status/index.html>
- [7] The Etherboot Project. <http://www.etherboot.org>
- [8] Ralf Brown's Interrupt List. <http://www2.cs.cmu.edu/~ralf/files.html>



# Console over Ethernet

Michael Kistler, Eric Van Hensbergen, and Freeman Rawson  
*IBM Austin Research Laboratory*

## Abstract

While console support is one of the most mundane portions of an operating system, console access is critical, especially during system debugging and for some low-level system-administration operations. As a result, densely packed servers and server appliances have either keyboard/video/mouse (KVM) cabling and KVM switches or serial ports, cabling, and concentrators. Further increases in the density of server appliances and blades require eliminating these items. We did so in the design of a prototype dense-server-blade system that includes none of the standard external ports for console devices. Since the standard means for console access, KVM or serial, were not possible, we developed low-level software to redirect console activity to the Ethernet interface. This paper describes the console support over the network interface that we developed for both our LinuxBIOS-based boot-time firmware and the Linux operating system. We refer to this code as *console over Ethernet* or *etherconsole*. We found it invaluable in debugging and evaluating our prototype server blades. We also describe ways of extending our work to make it more transparent to existing firmware and operating systems.

## 1 Introduction

Console support has been a feature of operating systems since their inception, and it is perhaps one of their most basic and mundane components. Modern computer systems make relatively little use of the console during normal operation and, for reasons of cost, space and simplicity are often run without dedicated terminal equipment connected to the console interface. This is particularly the case for servers packaged as server appliances, where a *server appliance* is defined to be a computer system specialized to run a single server application, such as a web server or firewall. However, during system debug and for certain types of low-level system-administration tasks, console support is essential.

In this paper we describe several variants of an implementation of Linux console support that use the Ethernet connection as the input/output device. We incorporated versions of this console support into both LinuxBIOS [18] and Linux to provide the console support for a prototype dense-server-blade system that includes none of the standard external ports for console devices.

We call our console support *console over Ethernet* or *etherconsole*. We found this code to be invaluable for both the hardware and software bring-up and ongoing systems management of our prototype. As we used our implementation, we refined and extended its usefulness, but we also identified some areas where further enhancements are clearly desirable.

Currently, there are some who believe that consoles are an anachronism, a hold-over from earlier days, and, clearly, there are embedded systems environments that have no console support at all. However, we believe that for servers, especially during certain portions of their life-cycles, console function is critical. As noted above, we needed console support during our debugging efforts, and we would note that even embedded systems developers generally use some type of temporary console support, usually some form of console over a serial line, during the debugging of their code. For servers, given the complexity and flexibility of their configurations and their tendency to have a repair-or-replace rather than a pure replacement strategy for dealing with failures, there is a need for console support beyond the development phase to set low-level configuration values and capture system message output. In the future, servers may become more like embedded systems and no longer require console support except during development, but that does not seem to be the current situation.

The following section provides some additional motivational background for our work as well as describing the constraints of our environment. Section 3 describes other console technologies and previous, related work. In Section 4 we give a detailed description of the design and implementation of *etherconsole*. Section 5 provides the results of our experimental evaluation of *etherconsole*, and Section 6 indicates a number of ways in which our work can be enhanced. Section 7 concludes the paper.

## 2 Motivation and Intended Environment

Our work on *etherconsole* is in support of two larger projects at the IBM Austin Research Laboratory, the MetaServer and Super-Dense Server projects [10]. The goal of the MetaServer project is to develop new approaches to systems management for clusters of server appliances. A MetaServer cluster consists of a collection of compute nodes, each of which contains a pro-

cessor and memory but no local disk, network-attached-storage nodes, which are used to store application data, and a MetaServer system that provides a single point of management and administration for the cluster. These components are connected by a high speed, switched network infrastructure, and the cluster connects to an external network through one or more gateways that hide its internal organization. The compute nodes share a single operating system kernel image and root file system supplied to them over NFS from the MetaServer system. The operating system environment of the compute nodes is based on the Red Hat 7.1 Linux distribution and the Linux 2.4.17 kernel, with key modifications to support diskless operation. We call this environment the *Linux Diskless Server Architecture* (Linux-DSA).

The Super-Dense Server (SDS) project is exploring ways of building servers to reduce their power consumption and heat production as well as increase their density. It uses a number of boards, or *server blades*, plugged into a chassis or backplane where each server blade is a single-board computer with a processor, memory, an Ethernet connection, an interface to the I2C system management bus, and supporting logic. The server blades were intentionally designed with a very limited amount of I/O to reduce their power consumption and heat production and to increase the overall density of the prototype. In particular, our server blades do not have local disks and do not include any of the standard external ports for console devices. Along with the server blades, other specialized blades are also plugged into the chassis including an Ethernet switch and a specialized single-board computer that acts as a system management controller. Each server blade runs its own copy of a common operating system image and server-application set, and the server blades in a chassis are typically configured into a cluster. We developed our own boot-time firmware based on LinuxBIOS [18] and use the Linux-DSA operating system and runtime environment from the MetaServer project. Recently, a number of commercial dense-server and server-blade products have appeared in the marketplace, including ones from Amplus [1], Compaq [6] (prior to its merger with Hewlett-Packard), Hewlett-Packard [12], IBM [14], and Dell [7]. All of these products share concepts similar to those found in the SDS work, but all still include either standard or proprietary ports for console devices and require extra cabling and switches for console access.

The goal of etherconsole is to attach a console or console emulator to a server over a standard Ethernet connection. In so doing, it provides a complete replacement for all console alternatives for both the firmware and the operating system. Since a server must have a network connection (typically Ethernet in commercial data center environments), reusing it for console attach-

ment adds no additional hardware or cabling for console support. The network interfaces, cables (or backplane wiring) and switches are all required and would be present no matter how the console is attached. Given our desire to maximize the packaging density of our server blades, which led to our decision not to include the ports required to support standard console interfaces, we were forced to develop an alternative mechanism for console support. Our only options were the Ethernet connection and the I2C system management bus. We rejected the I2C bus for a number of reasons. First, we wanted to reserve it for hardware control and certain low-level monitoring functions that we needed for our other research work. Second, although the latest versions of the bus have up to 3.4 megabits per second of bandwidth [23] in high-speed mode, our implementation runs it at the bus's original bandwidth of 100 kilobits per second. Third, experimentally, we found the I2C bus to be somewhat unreliable in our implementation. Once we eliminated the I2C bus from consideration, we were forced to implement the console support for our prototype over the Ethernet interface.

Our network environment also influenced our design. We made several important assumptions about the Ethernet and the systems on it, all of which are true in our environment and many other clusters and data centers. First, we assumed the use of high-speed, highly reliable links connecting the individual server systems to an Ethernet switch. Second, we assumed that the switch is capable of Ethernet data-link-layer [22] switching at line speed, which means that it does not experience congestion problems or drop packets. Therefore, the network itself is highly reliable, and only the system endpoints create transmission or reception hazards. Third, since all of the systems, including the management system that runs the console monitor or console emulator program, are on the same physical subnet, we assumed that no routing is needed. These assumptions imply that reasonably reliable, high-performance communication is possible using only link-layer Ethernet protocols and hardware addressing.

Since we were developing a prototype hardware platform, we had to do system debug and bring-up. We initially developed two forms of console over Ethernet, a link-layer, output-only one, and a full console implementation using the TCP/IP stack. Recently, we consolidated them into a single link-layer implementation that can function both as a simple message transmitter and as a full console supporting both input and output. We needed the link-layer, output-only function to allow us to track the progress (or lack of it) of early system initialization and the full-function support as a complete replacement for the standard console function. Full console support is required, for example, when some type

of low-level configuration is needed and the system must run in single-user mode. Although our systems have network access in single-user mode, standard daemon processes such as telnet servers cannot run, so the console is the only way of providing the input that the machine requires.

### 3 Background and Related Work

Others have recognized the cost and complexity issues with providing console access to large clusters of servers and have developed a variety of solutions to address these problems. In this section, we describe both the alternative approaches to console support used in dense servers and blades and some work that is a pre-cursor to our own solution to the problem.

#### 3.1 Alternative Approaches to Console Support

We have identified five distinct approaches to providing console support that are in common use in commercial data centers. The first approach employs a Keyboard/Video/Mouse (KVM) switch which connects a single keyboard, mouse, and display to multiple servers, typically 4, 8, 16, or 32 machines. To interact with the console of a particular server, the user selects the server using a mechanical or programmable selector on the KVM switch, and the keyboard, mouse, and display are connected by the switch to that server's console ports. A KVM switch that supports  $N$  servers requires  $3N + 3$  cables – 3 between each server and the KVM switch and 3 from the KVM switch to the shared keyboard, mouse, and display. In addition to standard KVM switches, there are a number of KVM extenders as well as KVM-to-Ethernet concentrators that have appeared on the market. One advantage of KVM switches and extenders is that they are capable of presenting a graphical user interface. However, in comparison to console over Ethernet, all of these products require additional hardware components, cabling, and space.

A common alternative to KVM switches is the use of a serial console which redirects the console traffic over a serial port connected to a serial concentrator. Unlike KVM switches but like etherconsole, serial console does not support a graphical user interface. Serial consoles and concentrators offer the advantage of reducing the number of cables for  $N$  systems to  $N + 1$ . Some newer serial concentrators also offer conversion to an Ethernet uplink. However, they still add to the cost of the installation, take up space, and require additional cabling. Given that the console is a vital but rarely used function in a production installation, this seems like an excessive price to pay for supporting it.

Recent developments in hardware, firmware, and operating system support now make it possible to use

a Universal Serial Bus (USB) [27] port for the console. The current support in Linux for a full console over USB is similar to one of our implementations in that it is based on the pre-existing serial console support in Linux and is not graphical in nature. The USB serial console supports communication with a terminal emulator over a USB-connected serial port. Like the serial console, the USB serial console requires at least  $N + 1$  cables for  $N$  systems as well as the additional serial concentration hardware and a USB-to-DB9-serial adaptor for every system. In addition, given the late initialization of the USB support and infrastructure in Linux, the USB serial console offers less support for system debug than the standard serial console. Moreover, USB cables are limited to 5 meters in length, complicating the cabling problem. Although there are also KVM switches that support the USB mouse and keyboard interfaces, they still use the standard video interface and cable and have properties that are very similar to standard KVM switching schemes.

IBM has recently introduced a new form of KVM switching, C2T Interconnect cable chaining and the NetBAY Advanced Connectivity Technology (ACT) [15]. Rather than using a star topology for KVM switching, ACT daisy chains the server systems together. C2T is one of the two chaining techniques used; it reduces the cabling and conversion hardware at the cost of adding a specialized port to every server in the chain. The daisy chain ends in either a special KVM switch or KVM-to-Ethernet converter. Although ACT reduces the amount of cabling and the number of switches or converters required, it still relies on cabling, connectors, and supporting equipment that reduce density and increase cost.

Finally, there are systems that provide a form of integrated, hardware-based console. A good example of such an implementation is the Embedded Remote Access (ERA) port [9] found on some recent server systems from Dell. The ERA port provides console access to firmware functions as well as hardware monitoring information using an independent microprocessor, memory, and set of firmware, and a dedicated network port, presumably an Ethernet port. Based on the published information, it appears that the ERA collects information over the I2C bus. It serves only as a hardware and firmware console, but not as the console for the operating system. It may be possible to modify the operating system to interact with the ERA over the I2C. This approach is probably adequate for a single system, but the characteristics of the I2C bus may well make it unsuitable for clusters.

In summary, all of these approaches require additional cabling or hardware, which decreases reliability and adds complexity to the installation and management of a cluster of server blades or appliances. In ad-



dition, they increase costs, in terms of equipment and floor space requirements, because they require additional hardware devices such as KVM switches or serial concentrators. Finally, all of these console implementations decrease the density of the server systems themselves since they require parts on the server boards that can be eliminated if console access is provided through the network interface.

## 3.2 Other Software Technologies

Portions of our implementation of console over Ethernet were based on the recently introduced Linux netconsole feature [17, 25]. After we describe our design and implementation, we will compare our work with netconsole in Section 5.4. Here we merely review the key features of netconsole. As implemented in the openly released patch to Linux 2.4, netconsole is a module that is dynamically loaded into a running kernel image. It is designed to support the transmission of console messages, especially in emergency situations, but it is not intended as a full console replacement. The netconsole implementation uses UDP and either broadcasts messages or transmits them to a particular receiving host, whose IP address is specified as a parameter to the netconsole kernel module when it is loaded. Netconsole registers as a console to allow it to receive the messages being written by the kernel. To avoid problems when the system is low on memory, netconsole maintains a dedicated pool of Linux `sk_buffs` [3], the network buffer structures used by the kernel. It also relies on special routines added to the network device drivers that send out frames waiting in the transmission queue without enabling interrupts, polling for the device to become available if necessary. Netconsole uses a syslog server running on another system to capture the messages that it sends.

In addition to netconsole, there have been other console-over-a-network implementations in recent years. The Fairisle network console [2] is similar to etherconsole in that it provides remote console support over a network, in this case, an ATM network. Based on the readily available documentation, it is difficult to make a detailed comparison of etherconsole with the Fairisle network console. In particular, it is not clear whether the Fairisle network console is a complete replacement for all local console function.

One traditional method of accessing systems is through terminal-oriented protocols such as telnet [5] and SSH [30]. Although these protocols provide tty access to the system and most of the needed function once the operating system and all of the required infrastructure are running, unlike etherconsole, they do not offer true console access and are not available until after kernel and multi-user system initialization has been completed.

Virtual Network Computing (VNC) [26] is a very commonly used way of gaining access to a graphical user interface (GUI) over the network. It is often used, for example, to provide an X Windows session with a remote Linux host on a Microsoft Windows-based system. Like telnet and SSH, VNC depends on a broad range of underlying system services and thus cannot be used during system initialization. VNC also requires more network bandwidth since it must transfer graphical screen contents to the remote system. Also unlike etherconsole, which is intended to work with server programs that catch messages from and interact with large numbers of server blades, VNC is intended to allow remote, one-to-one access to the graphical user interfaces of a relatively small number of host systems. Another commercially available remote GUI-access protocol is Citrix's Independent Computing Architecture (ICA) [4], which is used with Microsoft Windows-based hosts. Although the details of ICA are very different from those of VNC, they are conceptually quite similar [21].

Finally, although there is a SourceForge project entitled "Network Console Project" [20], as of this writing (early 2003), it appears to be dormant.

## 4 Design and Implementation

We developed three different console over Ethernet implementations, and each implementation contributed features that we found very useful in our work. The first implementation, which we call *etherconsole-T*, is a full replacement for the standard serial console. Etherconsole-T uses a kernel-level interface to the socket library to send and receive console messages on a TCP connection over the Ethernet instead of doing low-level device operations on the serial port as the serial console does. The second implementation, referred to in this paper as *coe*, is actually more primitive in terms of function and reuses some technology from the Linux netconsole implementation described in Section 3. Despite its simplicity, we found *coe* extremely useful in our work. The final implementation, called *etherconsole*, merges the best features of the two previous implementations. This section describes, in turn, each design and implementation along with the console emulator code that we developed to work with it.

### 4.1 TCP/IP-Based Implementation

Our initial implementation of console over Ethernet is a straightforward implementation of all of the console and tty function of the serial console using a TCP connection rather than a serial line as the transport. Structurally, as well as functionally, etherconsole-T is very similar to the serial tty and console code in the standard Linux distribution with the major difference being the way that the driver code communicates with the console emulation



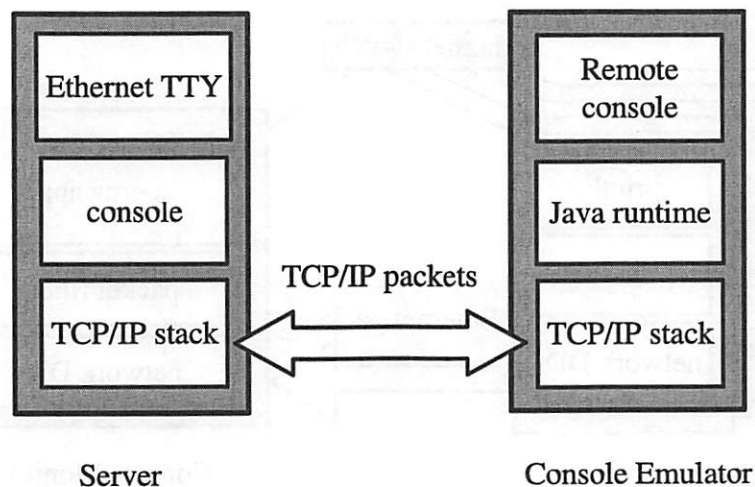


Figure 1: etherconsole-T: a TCP/IP-based console over Ethernet

program running on another system. Since etherconsole-T predated our other work with link-layer communication protocols, we chose to implement it using transport-layer protocols rather than link-layer networking. Figure 1 shows the overall structure of etherconsole-T and its console emulator program.

Since etherconsole-T is an Ethernet tty, it must fit into Linux's overall tty structure including the kernel line discipline code. It is structured as a character device with a major and minor number, relying on either `/dev` entries or the `devfs` [11] file system to provide the required special files. The driver supports multiple minor numbers and, therefore, multiple Ethernet ttys. The `console=` boot parameter is used to specify the name of the etherconsole-T device and the IP address of the system running the console emulator program. Since etherconsole-T depends on a fully configured TCP/IP stack, console initialization must be delayed until after TCP/IP is initialized and configured. If etherconsole-T is to be used for diagnosing early boot problems, TCP/IP configuration must be done as part of kernel initialization, using either DHCP or parameters passed to the kernel. Our server blades use DHCP to obtain the kernel boot image and configuration parameters and always perform TCP/IP configuration as part of kernel initialization. When etherconsole-T is initialized, it establishes two socket connections with the console emulator program on the remote host. One of the connections is used to accept input from the console emulator program while the other is used to write data to it. The etherconsole-T code does all of its communication with the console emulator using a kernel-level interface to the standard socket library; the programming technique it uses is very similar to the one used by the in-kernel

Tux [24] web server. Since input from the console emulator arrives asynchronously, etherconsole-T uses a dedicated kernel thread to read it and pass it up through the tty code.

In addition to the code that works with the tty line discipline function in the Linux kernel, there is also a set of direct I/O routines for use by the kernel console function. These operate in much the same manner as the functions that interface with the tty code except that they are invoked through the Linux console structure.

We modified a terminal emulator and telnet program written in Java [16] to act as a console emulator for etherconsole-T. The result uses standard telnet protocol and terminal emulation techniques once communication is initiated, but unlike a standard telnet client, it listens for connections from a system running etherconsole-T, and when it accepts a pair of connections (one input and one output) from a server, it opens a separate window for the console session with the remote system.

Etherconsole-T adds two new files to the Linux kernel sources with a combined size of 1243 lines of code. In addition, there are 12 lines of code added to the tty and system initialization. Our changes to the Java telnet program to support console emulation are about 220 lines of Java.

## 4.2 Link-Layer Console Message Transmitter

Although etherconsole-T works well in the context of our Linux-DSA operating environment, it requires a full Linux TCP/IP stack and, thus, cannot be used in very low-level code such as boot-time firmware. Our need for console support to help debug our server blades led to our second implementation, `coe`. Figure 2 depicts the

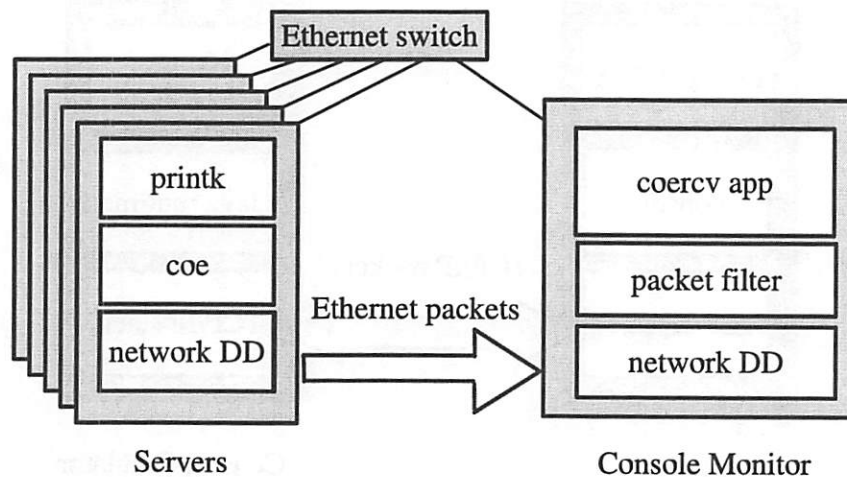


Figure 2: coe as deployed on prototype server blades

implementation of coe on our prototype server blades.

We began our work on coe by implementing some simple extensions to Etherboot [29] and LinuxBIOS which allowed redirection of print messages over the Ethernet device as opposed to a serial port. We then combined this code with portions of Red Hat's netconsole patch in order to provide a unified, link-layer, message transmitter for both our firmware and the Linux kernel.

Our implementation of coe uses link-layer networking only, transmitting raw Ethernet frames with a special frame type. This allows our console to begin operation much earlier in system initialization since it does not depend on IP configuration. Even when done at the kernel level, IP configuration occurs relatively late in initialization and relies on the proper exchange of messages between a kernel-level DHCP client and a DHCP server or the proper processing of kernel parameters. The special Ethernet frame type allows other systems to identify and process coe packets properly and ensures that these packets do not conflict with the existing traffic on the network. The use of link-layer networking also reduces the number of locks that must be obtained while formatting the message for transmission since the network stack's sequence number lock is no longer needed. To minimize the amount of configuration required and to eliminate the dependency on module parameters, we changed the code to broadcast Ethernet frames to the hardware broadcast address of `ff:ff:ff:ff:ff:ff`.

We retained from netconsole the use of a pool of reserved, pre-allocated network buffers that allows for the transmission of up to 32 console messages in out-of-memory situations in which the standard network buffer

allocation fails. We also used the same device polling technique for message transmission over the network. Network hardware generally has some limitation on the size of its transmission queue. The coe implementation checks to determine if the transmission queue of the network device is stopped. If so, it calls a routine in the network device driver that enters the interrupt handler to force the processing of transmission interrupts, thus clearing the queue and allowing coe to transmit a packet. This scheme improves the reliability and timeliness of the transmissions.

To receive and display console messages sent by coe, we wrote a console monitor program, `coercv`, that runs on a separate system in the same physical subnet-work. It uses the packet socket support provided in the standard Linux kernel to send and receive raw network packets. `Coercv` creates a packet socket using the standard `socket` system call, specifying `PF_PACKET` as the protocol family and the coe Ethernet frame type as the protocol. This causes the Linux kernel to set up a packet filter [19] for the coe Ethernet frame type; the packet filter directs incoming frames with the specified frame type to `coercv`. The `coercv` program uses the `recvfrom` system call to receive the incoming messages. For sockets in the `PF_PACKET` protocol family, the kernel returns the hardware address of the source of the packet in the `sock_addr` structure passed on the `recvfrom` call. This allows `coercv` to either tag the messages that it prints with the Ethernet hardware address or filter the incoming messages and print only those whose source hardware address matches a specified value. We also added support to the standard Linux `syslogd` program, which logs messages, to capture and log ones being sent by coe. It uses a map of network

hardware addresses to tag the messages that it receives with the hostname of the server to allow the reader of the log file to determine what system originated each message.

Neither `coe` nor `coercv` contains much code or is very complex. Building on the then-current version of `netconsole`, `coe` adds 35 lines of code to the source file, of which 26 lines are preprocessor conditional compilation statements. The `coercv` program with optional filtration based on the sending system is slightly more than 100 lines of code. The changes to `syslogd` are approximately 75 lines of code.

### 4.3 Merged Implementation

Finally, our `etherconsole` implementation combines the features of the `coe` and `etherconsole-T` implementations. It offers both the low-level, output-only broadcast of messages from the system and a complete console and `tty` similar in function to `etherconsole-T`. Although it incorporates the kernel module packaging features, it is designed to be built statically into a kernel image, as we must do with `LinuxBIOS`, and this also allows it to be initialized earlier than would be possible if it were a kernel module. Once the network device driver and basic sockets support have been initialized, `etherconsole` registers as a console and can start broadcasting messages. The Linux kernel maintains a buffer of the most recently issued console messages and passes these to the console during the registration process. This generally allows `etherconsole` to process all the messages generated by the system during the boot process, even those issued before the network is available. The size of the kernel message buffer can be expanded to prevent the loss of early boot messages because of buffer wrapping, but this has not been necessary in our environment. Although `etherconsole` initialization is later than standard console initialization, it still precedes most system initialization processing and, therefore, supports error diagnosis for a large fraction of system initialization.

`Etherconsole` uses a link-layer protocol and, thus, it can communicate only with other machines on the same physical subnet. In its default configuration, it performs all of its output by setting the Ethernet MAC address to `ff:ff:ff:ff:ff:ff` to force the hardware to do a broadcast. However, it can be configured to send output to a particular target MAC address with the `console=` kernel boot parameter. The syntax of this boot parameter is

```
console=ttyNn,mac_address
```

where `ttyNn` is a character-device node in `/dev` or created by `devfs` [11] with a major and minor number indicating it is an `etherconsole` device and `mac_address`

is an Ethernet MAC address in 6-byte, hexadecimal notation.

Packets are transmitted by queuing them directly on the transmission queue of the Ethernet driver and invoking a special routine in the device driver to send out available packets without enabling interrupts, polling for the device to become available if necessary. To handle input using link-layer protocols, `etherconsole` registers as the packet handler for its Ethernet frame type, so that it receives all incoming packets with that frame type. `Etherconsole` discriminates between packets that are console input and those that are broadcasts from other `etherconsoles` on the same subnet by checking the destination address in the Ethernet frame header. If it matches the hardware address of the local Ethernet interface, the packet is assumed to be console input. Otherwise, it is treated as a broadcast from another `etherconsole` and discarded.

Like `etherconsole-T`, `etherconsole` contains `tty` support as well, which allows us to use it as a full console replacement. To provide `tty` support, `etherconsole` requires the character-device node described above. Early in boot, the system uses the console entry points directly, but once the `init` thread opens `/dev/console`, subsequent console interactions use the `tty` entry points. `Etherconsole` provides all of the entry points needed by the `tty` code. However, many of the entry points do nothing more than return since they are not applicable to the underlying Ethernet device.

Our final `etherconsole` implementation supports `coercv` and the modified `syslogd` we developed for `coe` as well as a Java console emulator similar to that used with `etherconsole-T`.

Our link-layer `etherconsole` including both the code inherited from the original `netconsole` implementation and all of the function that we added is 470 lines of code.

## 5 Experience and Evaluation

This section describes our experience with our console-over-Ethernet implementations.

### 5.1 Usage

Since our server blades are prototypes, we wrote our own boot firmware based on `LinuxBIOS` [18] and `etherboot` [29], which allowed us to integrate `etherconsole` into both our firmware and our Linux kernels. Using `etherconsole` allowed us to debug and perform low-level systems management for server blades that do not have any standard external ports for console devices. We were able to track the progress of boot from power-on or reset through power-on self-test, firmware initialization, hardware set-up, and operating system load to the completion of kernel initialization. This proved invaluable



for fault isolation and problem determination. It also allowed us to monitor the concurrent initialization of multiple blades on a single display, making it easy to determine when the blades were all up following a cluster reboot. Moreover, it externalized kernel messages regarding problems encountered during normal operation. Our full console function allowed us to manage systems running in single-user mode, start and stop subsystems outside the context of normal init processing, and perform other types of management and debugging operations when normal telnet or SSH services were unavailable.

One usage difficulty that is not an issue in our environment, but might be in other situations, is the fact that there are a few commonly used commands such as `ifconfig` that manipulate the Ethernet interface and the network configuration. Since our systems are diskless and totally dependent on network communication, shutting down the Ethernet interface with an `ifconfig` down command effectively shuts down an entire node of our cluster. However, for disk-based environments, care must be taken to restrict the usage of commands that may unintentionally terminate console operation.

## 5.2 Performance and Reliability

Since console and tty support is not a performance path, we did not perform any formal performance studies. However, in our very extensive experience with our implementations, we did not observe any performance problems, in terms of either the speed of the console or the overhead imposed on the system. Also we found that we did not lose messages even when using the implementations that operate at the link layer. Even for a configuration where 8 blades are broadcasting boot-time messages simultaneously, the period of heaviest activity, the load imposed on a switched 100 Mbit Ethernet is insignificant. If one or more specific MAC addresses are used instead, the network load is even less. As mentioned in Section 2, our network environment is such that the network itself is reliable at the link layer, and the implementation techniques used in our link-layer code avoid the problems that can occur with network transmission on busy systems. We have not found any need for additional reliable delivery or flow control mechanisms. In passing, it is worth noting that we also use link-layer networking very successfully in our environment for remote disk access [28].

Having a second network interface and dedicating it to console support offers the advantages of clean traffic separation and the possibility of having a different network topology for console messages, but our experience indicates that these benefits are outweighed by the cost, complexity and additional space required for this design. Our implementation of console over Ethernet

achieves more than adequate performance and reliability with very little impact on other network traffic using the single Ethernet interface on our blades. The paper by Felter, *et al.*, [10] contains detailed performance studies of our server blades. All of these studies were conducted with one or more of our console-over-Ethernet implementations active on all the blades.

## 5.3 Security Considerations

One possible concern with console over Ethernet is security. Putting the system console on the network creates the possibility of break-ins and denial-of-service attacks. By using link-layer protocols, etherconsole hides the console from systems outside the physical subnet. Link-layer protocols do not use IP addresses and are not routable beyond the subnet. In addition, the hardware addresses required for interaction with the console are not exposed and have no meaning outside the subnet. Within the subnet, access to the console can be further restricted using VLANs [13]. A private VLAN can be created consisting of only those machines that require console access; machines in the physical subnet but not part of the VLAN will be unable to access consoles within the VLAN. A more sophisticated console emulation program can act as a form of firewall and bridge, receiving connections from the outside on another VLAN, doing the necessary protocol conversions, and forwarding legitimate messages to the etherconsoles over the private VLAN. This idea is explored further in Section 6.

Denial-of-service attacks are a special form of security exposure that have recently received considerable attention. One possible form of this type of attack is to send the console a very large number of messages or faulty command strings to handle. Processing them can consume a substantial amount of resource, slowing the system down significantly. A related form of attack is to send a command that somehow causes a very large amount of console output. As most software developers recognize, one way to slow down a program dramatically is to have it do a very large number of `printf` calls, and one way to slow down a kernel module in Linux is to fill it with `printks`. In many cases, Linux protects against this form of attack by limiting message generation rates. For example, the rate limiting support in the TCP/IP protocol stack of the Linux 2.4 kernel detects and suppresses duplicate messages issued faster than some fixed rate. This decreases the load imposed by any console implementation, including etherconsole, and reduces the possibility of lost messages.

## 5.4 Comparison with Netconsole

We made three very significant extensions to netconsole. First, we designed etherconsole to be built into the ker-



nel and enabled at boot time, rather than being packaged as a kernel module that can be used only after the system is fully initialized. This allows etherconsole to start operation much sooner than netconsole, so that messages generated during system boot can be observed remotely. Second, etherconsole uses a link-layer protocol that requires only basic network hardware support and MAC-level addressing, whereas netconsole uses UDP sockets, requiring full IP configuration and addressing. Third, we added support for input by incorporating the tty support described above. In addition, etherconsole also derives from our very first implementation of the full console over Ethernet, which was a completely independent development that started before netconsole appeared.

## 6 Future Work

There are a number of areas in which our work can be extended, something that we hope to encourage by making our code available to the open source community.

### 6.1 Functional Extensions

Our implementation has a few functional deficiencies. In particular, it does not handle the `control-alt-delete` key sequence for rebooting nor does it process the `sys req` key used by some Linux kernel debugging features. Rather than relying on command line parameters, the DHCP [8] support in the firmware and kernel can be used to receive additional console configuration and control information such as the hardware address of a particular workstation running the console emulation program for this system.

An interesting and more challenging functional extension is to support a remote gdb debugger over the Ethernet. In principle, many of the same techniques used in the private gdb serial code should also work with an Ethernet interface. In particular, our code already uses polling to clear the transmission queue and operates with minimal overhead at the link layer. The major open question is how to handle the network buffers. The debugging environment is even more restrictive than the console, and we need to recycle a fixed pool of network buffers that is completely private to the gdb stub code to avoid unwanted interactions with the system being debugged.

### 6.2 Subnet Console Server and Reflector

A useful extension to our console emulator software would be to incorporate a telnet or SSH server that would allow console access outside the physical subnet. Such a program can accept telnet or SSH connections over TCP/IP from workstations outside the subnet, perform appropriate authorization and authentication checking, and then link these connections with the

console of a particular system, doing the conversion between the TCP/IP and link-layer protocols.

### 6.3 Serial Port Emulation

Most server firmware and operating systems allow console interactions to be directed to one of the computer's serial ports. As discussed above, this is commonly used as the means of accessing the console of a server appliance or blade. For systems using x86-compatible processors, the serial port is an I/O port with an address defined in the system configuration. The firmware or operating system uses the OUTB instruction to send data and commands to the serial port and the INB instruction to receive data or read the port status. An alternative console over Ethernet design can intercept data and commands issued to the serial port and redirect these over the network to a console emulator running on another system. In this solution, no change is necessary to the operating system; it is simply configured to use the serial console. Figure 3 illustrates this concept. As indicated in the figure, communication can be at either the link layer using Ethernet frames or the transport layer using TCP or UDP.

A variety of mechanisms can be used to intercept read and write requests to a serial port. One approach is to add this feature to the firmware of the machine. In this case, the firmware is configured to trap INB and OUTB instructions written to a specific port and process them on its own. Since machines capable of DHCP and remote boot already have the basic mechanisms for IP connectivity built into their firmware, we can leverage this existing support to provide the transport for console communications. Another approach is to develop special serial port hardware that, instead of transmitting data to a physical port on the machine, transmits it to the network interface. Finally, a programmable network interface can pretend to be the serial device by owning the bus resources for it. It would then translate operations on the serial I/O port into the corresponding network interactions.

### 6.4 Frame Buffer Emulation

Some operating systems do not support character-based consoles and require the use of a graphical user interface instead. They are better supported with a frame buffer emulation approach. In this case, an area of memory is used to hold a dummy frame buffer, and the network interface sends characters from the frame buffer. Input occurs by transmitting characters and mouse movements to the system through the network interface and then passing them to the operating system by emulating normal keyboard and mouse input.

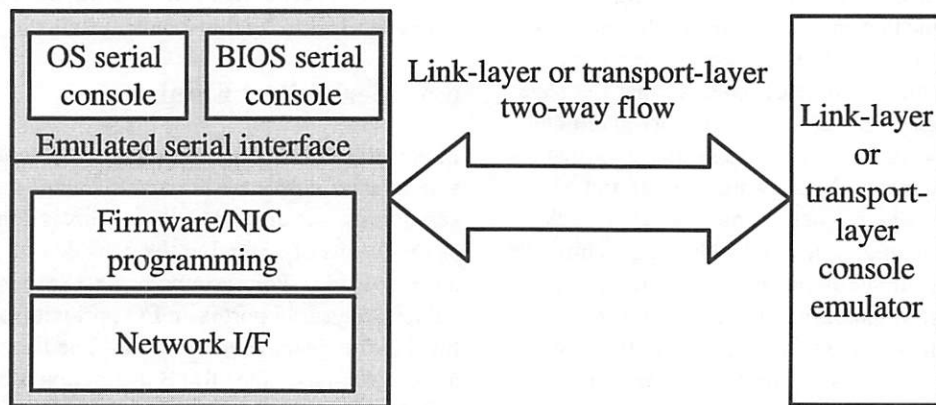


Figure 3: Serial device emulation

## 7 Conclusions

By redirecting console interactions over the network interface, console over Ethernet allows us to construct servers without standard external ports or cables and the associated infrastructure for console devices. This eliminates the cost, complexity, and fragility of console cabling and switches, while allowing for higher density packaging of server systems. As with the chassis that we used for our prototype server blades [10], even the Ethernet cables become unnecessary if the server is a blade that plugs into an appropriately wired backplane. Eliminating the separate network for console traffic simplifies the design and wiring of the backplane, reducing the cost of the chassis and the total cost of a blade server deployment.

## Source Code Availability

Source code for our implementation is available from the IBM Linux Technology Center web site at <http://oss.software.ibm.com/developerworks/opensource/linux>.

## Acknowledgments

Wes Felter, Tom Keller, Elmootazbellah Elnozahy, Bruce Smith, Ram Rajamony, Karthick Rajamani, and Charles Lefurgy contributed very heavily to the SDS project. We would also like to thank the management of the IBM Austin Research Laboratory for their support of our work. Our shepherd, Chuck Cranor, and the anonymous referees provided many helpful comments that improved the quality of this paper. All trademarks are the property of their respective owners.

## References

- [1] Amphus, Inc. Virgo: A ManageSite-enabled, fully manufacturable, ultra-dense server design, 2001.
- [2] R. Black. The Fairisle network console. <http://www.cl.cam.ac.uk/Research/SRG/bluebook/18/netcon/netcon.html>.
- [3] D. Bovet and M. Cesati. *Understanding the Linux Kernel, Second Edition*. O'Reilly and Associates, Inc., 2002.
- [4] Citrix Corporation. Citrix independent computing architecture. <http://www.citrix.com/press/corpinfo/ica.asp>, 2002.
- [5] D. Comer and D. Stevens. *Internetworking with TCP/IP, Volume 3: Client-Server Programming and Applications, BSD Socket Edition*. Prentice Hall, 1993.
- [6] Compaq, Inc. Proliant BL10e Server, January 2002.
- [7] J. Delaney. Dell blade: Maximum power, minimum space. *PC Magazine*, February 2003.
- [8] R. Droms and T. Lemon. *The DHCP Handbook: Understanding, Deploying, and Managing Automated Configuration Services*. Macmillan Technical Publishing, 1999.
- [9] Y.-C. Fang, J. Jancic, A. Saify, and S. Zaiback. Using Dell embedded remote access to facilitate remote management of HPC clusters. *Dell PowerSolutions*, November 2002.
- [10] W. Felter, T. Keller, M. Kistler, C. Lefurgy, K. Rajamani, R. Rajamony, F. Rawson, B. Smith, and E. VanHensbergen. On the performance and use of dense servers. To appear in the *IBM Journal of Research and Development*, 2003.

- [11] R. Gooch. Linux devfs (device file system) FAQ. <http://www.atnf.csiro.au/people/rgooch/linux/docs/devfs.html>, August 2002.
- [12] Hewlett-Packard Company. HP bc1100, December 2001.
- [13] IEEE. IEEE standards for local and metropolitan area networks: Virtual bridge local area networks, IEEE Standard 802.1Q-1998, 1998.
- [14] International Business Machines Corporation. BladeCenter. <http://www.ibm.com>, September 2002.
- [15] International Business Machines Corporation. The Decision Maker's Guide to Server Connectivity and Console Switching. <http://www.ibm.com>, July 2002.
- [16] The Java telnet/ssh applet. <http://javassh.org>, 2002.
- [17] M. Johnson. Red Hat, Inc.'s network console and crash dump facility. <http://www.redhat.com/support/wpapers/redhat/netdump>, 2002.
- [18] R. Minnich, J. Hendricks, and D. Webster. The Linux BIOS. In *The Fourth Annual Linux Showcase and Conference*, October 2000.
- [19] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 39–51, 1987.
- [20] Network Console Project. <http://sourceforge.net/projects/netconsole>, 2003.
- [21] J. Nieh, J. Yang, and N. Novik. A comparison of thin-client computing architectures. Technical Report CU-CS-022-00, Columbia University, November 2000.
- [22] L. Peterson and B. Davie. *Computer Networks, Second Edition*. Morgan Kaufman, 2000.
- [23] Philips Corporation. I2C-Bus. <http://www.semiconductors.philips.com/buses/i2c>, 2003.
- [24] Red Hat, Inc. TUX 2.1, 2001.
- [25] Red Hat, Inc. Netconsole, 2002.
- [26] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1), January/February 1998.
- [27] USB. <http://www.usb.org>, 2003.
- [28] E. Van Hensbergen and F. Rawson. Revisiting link-layer storage networking. Technical Report RC22609, IBM Research, October 2002.
- [29] K. Yap and M. Gutschke. Etherboot user manual, July 2002.
- [30] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH protocol architecture, September 2002.





# Implementing a Clonable Network Stack in the FreeBSD Kernel

Marko Zec

*University of Zagreb, Faculty of Electrical Engineering and Computing*  
zec@tel.fer.hr

## Abstract

Traditionally, UNIX operating systems have been equipped with monolithic network stack implementations, meaning all user processes have to cooperatively share a single networking subsystem. The introduction of the network stack cloning model enables the kernel to simultaneously maintain multiple independent and isolated network stack instances. Combined with forcible binding of user processes to individual network stacks, this concept can bring us a step closer to an efficient pseudo virtual machine functionality which opens new possibilities particularly in virtual hosting applications, as well as in other less obvious areas such as network simulation and advanced VPN provisioning. This article is focused on design, implementation and performance aspects of experimental clonable network stack support in the FreeBSD kernel.

## 1 Introduction

Implementing various models and levels of resource partitioning and protection has been in focus of operating systems research ever since the introduction of the multi-programming paradigm in early days of computing.

In the 1960s, IBM introduced the concept of virtual machines (VM). Each VM instance presented a close yet independent replica of the underlying physical machine which gave users the illusion of running their programs directly on the real hardware. VM also provided benefits like mutual isolation, protection and resource sharing, as well as the ability to run multiple independent flavors and configurations of operating systems (OS) simultaneously on the same physical machine.

On UNIX systems, user programs run in a simplified VM model. The OS kernel manages the allocation of protected virtual memory and schedules CPU cycles to user processes; however, the processes are not allowed to access any other hardware resources directly. Instead, the kernel provides an abstraction layer for accessing vital system resources, such as I/O devices, filesystems and network communication facilities. Traditionally, networking facilities in particular have been implemented monolithically within the kernel, meaning all the user processes have to cooperatively share the networking facilities and addressing space. Over the years this concept was completely sufficient for most of

the common environments; however it presents some limits to certain emerging applications, most notably to more sophisticated scenarios for virtual hosting.

The model of clonable network stacks, presented in this article, allows multiple independent network stack instances to coexist within the OS kernel simultaneously. From the perspective of network communications, by associating groups of user processes to an individual network stack instance it is possible to achieve highly efficient *light* or *pseudo* virtual machine functionality. Each unique collection of network stack instance and group of associated user processes will further be referenced as a *virtual image*.

The fundamental difference between the traditional VM implementation and pseudo-VM or virtual image model is illustrated in Figure 1. In both cases each VM appears as an independent entity from the perspective of the outside network. However, in the traditional VM system each VM hosts an entire OS instance with dedicated partitions of hardware resources such as physical RAM, raw disks and I/O devices. All I/O operations, including those dealing with network interfaces, have to be either controlled or emulated by the VM monitor. Contrary to this approach, in a pseudo-VM system, a single OS running on physical hardware controls multiple independent network stack instances and associated user processes without the need to dedicate hardware resources to each pseudo-VM instance. This allows for significantly less overhead on the data path from applications to the physical

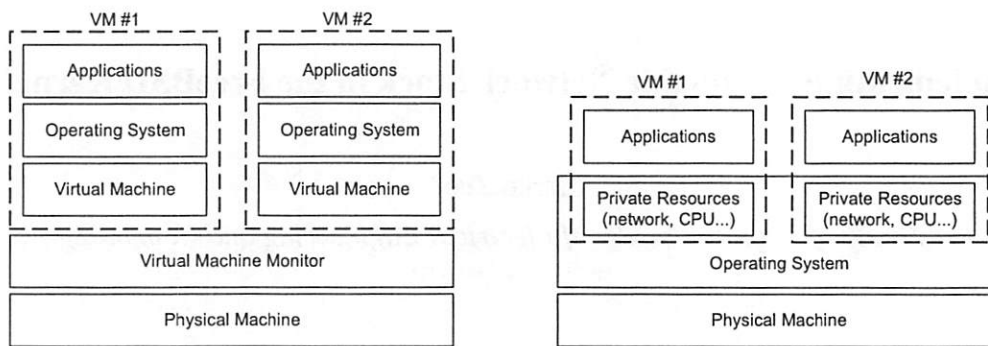


Figure 1: Traditional VM model (left) and pseudo-VM *virtual image* model (right)

network interfaces, as well as better utilization of other system resources, particularly CPU and RAM. On the other hand, the pseudo-VM system cannot run different OS flavors simultaneously and does not offer the level of mutual independence and protection available in the traditional VM model.

Although the described concept of virtual images and clonable network stacks is generic, this article is focused on its implementation in the FreeBSD kernel. With reasonable efforts and enough time, it should be possible to modify and extend other network stack implementations to offer such functionality, provided that the entire OS kernel source tree is available.

The rest of the article is organized as follows. Section 2 describes some previous work and models in partitioning system resources, with focus on network communication. Section 3 presents the basic design ideas behind the concept of network stack cloning. The implementation details are discussed in Section 4, followed by an overview of various performance implications presented in Section 5. Section 6 gives an example of how the management interface of clonable network stacks looks to the system administrator. Finally, Section 7 summarizes the properties of the clonable network stack infrastructure and outlines the directions for possible future development and research.

## 2 Previous work

The goal of running multiple instances of network protocol suite on a single physical machine can be accomplished in different ways. As mentioned in the previous section, one option is virtualization of the entire machine hardware and running multiple independent instances of fully self-contained OS images within the VM. Such an approach requires hardware virtualization support for efficient operation, which can usually be found only on large-scale and expensive

system architectures, such as IBM's S/390 mainframe family. Software-based virtualization of hardware architectures lacking the inherent virtualization circuitry is also achievable. Such virtualization products are available both as OSS and commercial products, such as VMware [1]. However, this model introduces a significant bottleneck at emulating I/O operations in software, which results in performance degradation during data transfers to and from the network interfaces. Therefore, often this approach may not be suitable for heavy-duty network centric applications. Furthermore, the traditional VM model can lead to suboptimal use of other hardware resources; for instance, the VM monitor is typically unaware if an OS instance running within a VM no longer needs a particular memory page.

An alternative model, the *jail* [2] facility implemented in FreeBSD, provides the ability to partition the OS into multiple separated process groups with limited network addressing space. The kernel prevents user processes running in jailed environments from managing the processes and certain system resources outside their own jailed protection domain. All the jailed environments share the same network stack; however each jail is restricted to use a unique IP address, and cannot interfere with other network traffic. Creating *jailed* pseudo virtual machines in this manner has many potential uses; thus far the most popular one has been for providing highly efficient virtual machine services in Internet Service Provider environments. It should be noted that the standard *jail* architecture still uses a monolithic network stack. Therefore the *jails* do not maintain private instances of subsystems such as routing tables, traffic counters, packet filters and traffic shapers etc., so they must rely on the *master* OS environment to manage those facilities.

Several reports describe efforts to implement network stacks as specialized userland processes, with the focus in network simulation applications. Frameworks such as ENTRAPID [7] or Alpine [8] successfully accomplish

their primary goals in network simulation, however at the cost of poor overall performance compared to the in-the-kernel network stack implementation on the same hardware. The Harvard network simulator [9] takes a different approach as it creates the illusion of having multiple independent kernel routing tables by providing transparent IP address remapping between user and kernel space. While the kernel still maintains a single routing table with unique (non-overlapping) entries, for each virtual node a translation map is maintained which has to be consulted on each userland-to-kernel network transaction. Despite such an approach offering notably better performance than the ENTRAPID and Alpine architectures, it is still significantly constrained by numerous translation lookups that have to be performed on each kernel-to-userland packet transition, and vice versa. However, a major advantage of the Harvard architecture over the other two mentioned simulator frameworks is the ability to use the existing UNIX network applications in a virtualized environment without any modifications.

Although none of the above network simulation frameworks have been designed for use in general-purpose or virtual hosting applications, some of their concepts and objectives were partially adopted in the implementation of the clonable network stacks model. This is particularly true for the idea of preserving full compatibility with the existing userland binaries running on a modified kernel, which was one of the important design goals in the experimental clonable network stack implementation.

### 3 Design concepts

By setting the goal to implement a system that would perform equally well in generic and virtual hosting, VPN packet routing/switching or network simulation applications, the idea was born to reuse an existing reliable but monolithic network stack implementation and extend it to support cloning in multiple independent and isolated instances.

**To virtualize the entire network stack** of the base OS, and not just the selected portions needed for certain applications, was the most important design decision. Further, it was decided to implement all the modifications entirely within the OS kernel. Such an implementation not only allows the new code to be highly efficient, but also provides a significant level of security and isolation between the virtualized environments needed in any virtual hosting application. If the virtualization extensions would be even partially implemented in userland, for example by replacing the

standard system dynamic libraries, it would be left up to the userland programs whether or not they would comply with their designation in a virtualized network stack. Another problem that would arise by choosing such an approach is that it could only work transparently for dynamically linked programs. Statically linked programs would have to be recompiled, which would clearly violate the requirement for transparent API/ABI compatibility with the user programs.

**Preserving the complete functionality of the base OS** serving as a development platform, while making the network stack extensions as universal as possible, was another important design goal. It became apparent that it would be highly desirable to preserve complete application programming and binary interface (API/ABI) compatibility with the existing application and utility programs running on the base OS. Such an approach would ensure simplicity and speed in both code development and testing, as well as in later real-world applications. If the API were not preserved, the end result would probably be a highly specialized system tuned for a particular application, not a general-purpose one as desired. Another important objective was the preservation of the performance of the existing base system. Obviously, if the modified system noticeably underperforms the original it could probably still be useful for certain specific applications; however, in such case the general-purpose criteria could not be met.

**The introduction of virtual images** was a key concept on which the clonable network stack framework is based. Each virtual image is an isolated kernel entity with its own independent network stack instance. All network interfaces and all user processes in a virtualized system are supposed to be associated with a unique virtual image, as shown in Figure 2.

Upon system startup only one (default) virtual image exists in the system, which contains all the network interfaces and user processes. System administrators can later dynamically create new virtual images and associate real or pseudo network interfaces with them and run user processes in those environments. User processes running in one virtual image will be able to interact only with their own network stack instance, therefore only with the network traffic and interfaces that are associated with its own virtual image. The default virtual image is no exception to this rule. As the virtual images are logically organized in a hierarchical manner, the parent virtual image is allowed to spawn a new process in its child for management purposes. Obviously, the child is prohibited from managing its siblings and the parent virtual image.

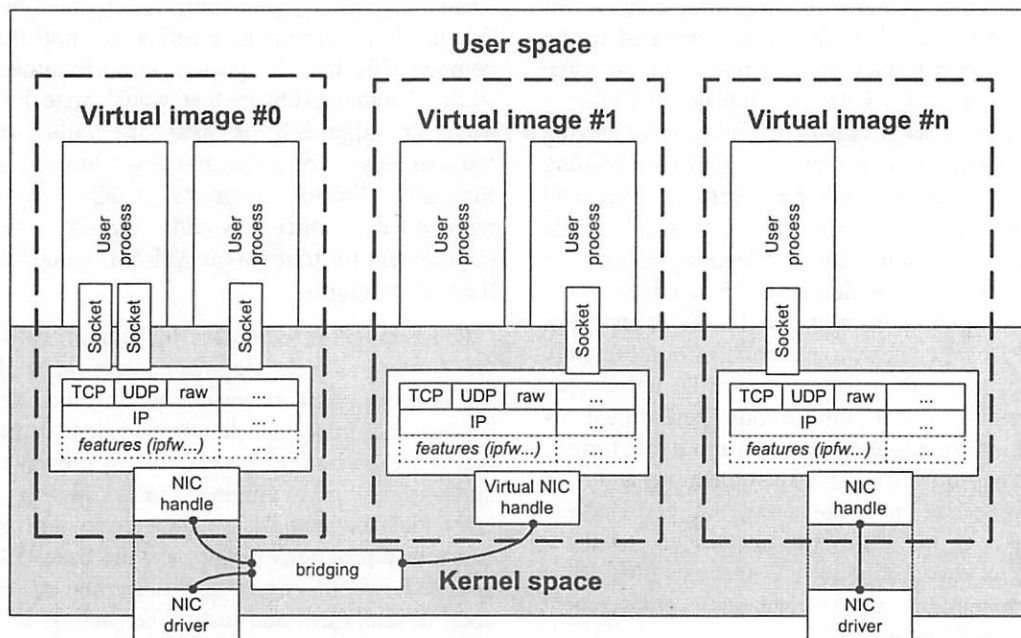


Figure 2: Operating system partitioned in virtual images

Although the high-level goal was to make virtual images as isolated and independent from each other as possible, they should be able to be interconnected if desired. This can be accomplished either via bridged virtual or physical Ethernet interfaces, or through virtual point-to-point channels constructed using the *netgraph* [10] framework. The former *bridging* method allows virtual images to become accessible from the outer world in a transparent manner, with almost neglectable overhead introduced by bridging code logic and processing.

## 4 Implementation

The experimental framework allowing multiple network stacks to be simultaneously active on a system was implemented as an extension to the FreeBSD 4.8 kernel. Each network stack instance was made fully independent of all others, so that each instance maintained its own private routing table, set of communication sockets and associated protocol control blocks etc. Later, the network stack virtualization experiment was extended to include optional networking facilities, such as packet filters, traffic shapers, bridging code and various `sysctl` [3] tunable variables controlling different aspects of network stack behavior.

### 4.1 Overview

As UNIX systems traditionally maintain only a single network stack within the kernel, an important design step has been selecting the optimal method for user processes to manage multiple network stacks. One option was modification of the standard Berkeley socket interface [4] by extending the argument lists with the network stack identifier. A variation of such approach was proposed in [11]. However, this concept has a significant drawback since it requires the existing user programs to be modified and recompiled in order to be able to run on the new / extended OS kernel. Therefore, an alternative approach was chosen. Each process control block (`struct proc`) in the kernel was transparently extended with a tag which associates it with a network stack instance. This tag is inherited by subsequent processes from its parents without any need for intervention from the programmer. Additionally, a new programming interface was introduced allowing a process to change its network stack association. This approach allowed for complete application programming and binary interface (API / ABI) compatibility to be preserved between the original and modified OS kernel, thus mitigating any need for modifications in the existing userland applications or utilities.

The described tagging of user processes was combined with the already available *jail* resource protection framework in FreeBSD, which resulted in



user processes associated with one network stack being effectively invisible to the other processes running on the system, and vice versa. The newly developed framework, which combined different areas of resource protection mechanisms into one entity, successfully achieved the main properties of pseudo virtual machine functionality. The *virtual image* concept was further extended by including modifications to the CPU scheduler in such a way that each virtual image could be limited in average CPU usage so that runaway or maliciously constructed processes or groups of processes might be prevented from monopolizing and starving all the real CPU resources. This also allowed system load monitoring to be performed on a per virtual image basis, which provided more fine-grained control rather than accounting resource usage solely on physical machine level. Finally, a basic API for managing the virtual images was implemented, accompanied by a simple userland management utility.

The fundamental approach taken in implementation of the described modifications to the FreeBSD kernel was the introduction of a new *vimage* kernel structure, which serves as a container for all virtualized variables and symbols. Gradually, most of the global and static symbols used by network stack code were replaced by their equivalent counterparts residing in independent

*vimage* structures. Network interface descriptors, which have traditionally been maintained in a single linked list, are now associated with *vimage* structures so that each network stack instance has its own list of network interfaces. Each network interface contains a pointer back to its *vimage* structure so that incoming traffic can be easily demultiplexed to the appropriate network stack depending on the interface the traffic is received on. A basic schematic diagram outlining the relationships between most important kernel structures in the clonable network stack implementation is shown in Figure 3.

## 4.2 Operation

As the symbols and data structures used for network processing, previously declared as global, are now placed inside the *vimage* structure, each function dealing with network traffic necessarily needs to know on which network stack it must operate. Typically, all the references to old symbols such as:

```
ip_ttl = ip_defttl;
```

had to be replaced with constructions such as:

```
struct vimage *vip = {current_vimage};
ip_ttl = vip->ip_defttl;
```

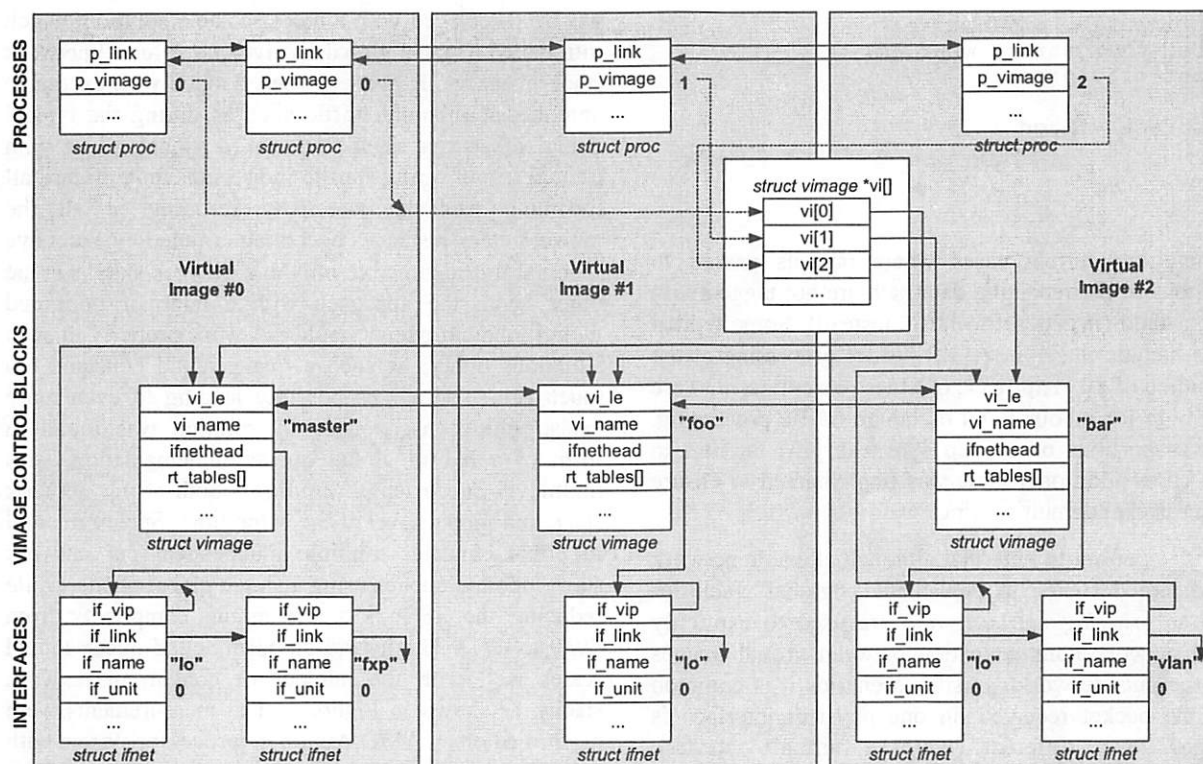


Figure 3 : Major kernel data structures separated and linked throughout *virtual images*

Replacing *all* occurrences of the relevant global variables by their virtualized counterparts throughout the kernel source tree was a textbook example of a time consuming job. After some unsuccessful attempts at automation this task was performed almost entirely manually. The initial patch against 4.7-RELEASE kernel source tree contained 5053 either modified or new lines of code in total.

**Basic operations** on network traffic are triggered by three types of events: arrival of packets from network interfaces, requests from user programs, and expiration of various timers.

**The incoming traffic** has to be demultiplexed to the appropriate network stack instance. The network stack instance is determined based on the interface the packet is received on, which can be easily accomplished since network interface descriptors have been extended to hold a pointer back to its *vimage* structure. In the 4.4BSD networking code, each packet is stored in specialized memory structures called *mbuf* [4]. A field in an *mbuf* header is dedicated for holding a pointer to the ingress interface for received traffic. Therefore, the functions that process inbound traffic can extract the information on network stack association for each received frame using the following or similar code (example):

```
void
icmp_input(m, off, proto)
    register struct mbuf *m;
    int off, proto;
{
    struct vimage *vip =
        m->m_pkthdr.rcvif->if_vip;
    ...
}
```

There are certain cases where packets passed to functions in the receiving data path are not tagged with *rcvif* field (it gets set to NULL instead). Some typical cases include *dumynet* [12] header processing at the beginning of *ip\_input()*, or other subroutines that are used both for inbound and outbound traffic processing, such as portions of the *tcp\_syncache* [13] facility. In such cases additional logic was implemented to ensure proper packet demultiplexing.

It is important to note that although clonable network stacks are designed to be isolated, "global" facilities such as *bridging* and *netgraph* are used to explicitly allow network communication between virtual images and the outside world. In such scenarios it is common that the packet received on one physical interface is bridged to a different (typically virtual) interface residing in another network stack. Therefore the bridging code and similar multiplexers must provide

proper retagging of the *rcvif* field in the *mbuf* header when passing the packets in the upstream direction.

The standard 4.4BSD model for processing incoming network traffic is split into two stages [4], resembling the concept of network protocol layering. During the first stage, which deals with *data link* layer, the received frames are demultiplexed to specific network protocol queues (IP, IPX...) and a software interrupt is scheduled by calling *schednetisr()* for the appropriate protocol handler. After all received packets are enqueued, the protocol-level processing is performed in a *netisr()* loop, until all protocol-specific receive queues are completely flushed. While this model works more or less seamlessly in the original monolithic network stack implementation, in the clonable stack framework it does not scale well with large numbers of network stack instances. The problem is that during *netisr()* processing the independent inbound queues of *all* network stacks would have to be checked for pending packets, which is a task with complexity of  $O(N)$ , where  $N$  denotes the number of network stack instances present in the system. It is apparent that most of the checking would be completely unnecessary, since it can be expected that only a small number of network stacks are active at the same time.

Therefore the described model of linear traversing through all network stacks during *netisr()* processing has been replaced with a more scalable solution, which introduced a single global receive queue for all network stack instances. However, when flooded with excessive amounts of inbound traffic such as during the typical denial of service (DoS) attacks, a global queue with limited length would start to indiscriminately discard all incoming packets, potentially resulting in all the network stack instances becoming crippled by excessive inbound traffic aimed to only one network stack. On the other hand, the approach with a huge or unlimited global inbound queue would not work properly in such situations either; as such a queue could consume too much *mbuf* resources possibly leading to even more catastrophic consequences. A solution was found in form of a hybrid global queue implementation, with multilevel queue length limiting - both at per network stack and global level at the same time. Such a method in effect emulates multiple independent per network stack inbound queues using a single global queue, while reducing the *netisr()* processing complexity from  $O(N)$  to  $O(1)$ . The hybrid global inbound queue model scales well with the number of concurrent network stacks, as shown in Figure 4. The measurements were performed on an AMD Athlon uniprocessor system with a CPU clock of 1200 MHz, a bus clock of 100 MHz, 256 Mbytes of SDRAM, and two Intel 82558B fast Ethernet cards connected to a 32-bit 33 MHz PCI bus.

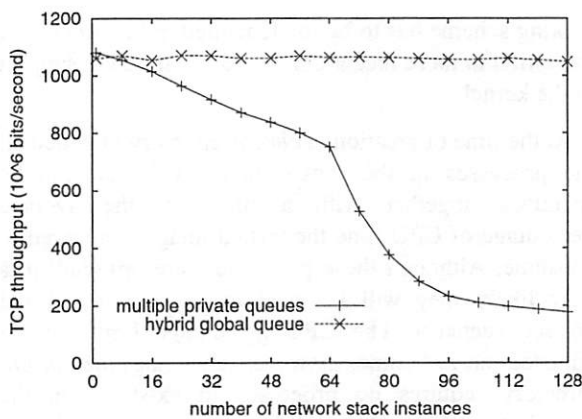


Figure 4: loopback TCP throughput in one network stack instance – comparison between linear traversing through independent per network stack queues vs. single hybrid global queue. MTU size is 1500 bytes.

The same machine also served as a referent platform in all other measurements presented further in text.

**User requests** are treated differently depending whether they perform socket operations or other tasks. At the time of creation, each socket is associated with a network stack instance that the owner process is currently bound to, using the newly introduced `so_vip` pointer in `struct socket` descriptor. For the whole duration of its lifetime, the socket remains tied to this network stack instance, even if the owning process changes the network stack association. The functions operating on sockets will therefore determine the network stack instance (or more precisely the *virtual image*) to work with similarly to the following example:

```
static int
tcp_usr_send(struct socket *so, ...)
{
    struct vimage *vip = so->so_vip;
    ...
}
```

Non-socket operations use the `p_vimage` tag in `struct proc` of the calling process to determine the stack instance on which to operate:

```
static int
rip_pcblist(SYSCTL_HANDLER_ARGS)
{
    struct proc *p = req->p;
    struct vimage *vip = vi[p->p_vimage];
    ...
}
```

In the current implementation the `p_vimage` tag is a 16-bit index to an auxiliary array of pointers `struct vimage *vi[]`. Preserving ABI compatibility with userland was the only reason for such a design, as no spare room for holding a direct pointer to `struct`

`vimage` in `struct proc` was available. Retaining both the same size and structure of `struct proc` in the modified kernel was a prerequisite for userland utilities such as `ps` or `top` to operate correctly without recompiling. In the future, the `vi[]` auxiliary array is expected to become obsolete.

**Timeout operations**, which are typically associated with subsystems such as TCP and ARP processing or *dumynet* [12] delay queues, have to be performed periodically. Those tasks are implemented as linear traversing through all network stack instances and calling the `{slowtimo|fasttimo}` handlers in each supported protocol suite, with pointer to the appropriate `struct vimage` as the argument. Since such events occur synchronously and with significantly smaller frequency than the reception of incoming traffic (only a couple of times per second typically), polling all network stack instances on each invocation of timeout processing subroutines generally cannot have noticeable influence on the overall system performance.

Besides those "core" network stack components described above, some other standard userland-to-kernel interfaces had to be adjusted and extended to support clonable network stacks.

The **sysctl** interface was originally designed to allow system administrators to conveniently monitor and adjust tunable parameters controlling different portions of OS behavior, including the network stack. As the standard `SYSCTL` primitives operate with symbols defined globally within the kernel, they were not directly suitable for accessing the symbols "hidden" inside a `struct vimage`. Therefore a collection of new primitives was introduced. For example the `SYSCTL_V_INT()` macro was implemented for allowing the access to integer variables within a `struct vimage`, replacing the original macro `SYSCTL_INT()` aimed for accessing *global* integer symbols. The `SYSCTL_V` family of macros determines the virtual image they operate on based on the `p_vimage` tag of the calling process.

The **kernel symbol lookup** interface (`kldsym / kvm_nlist`) is used by userland utilities such as `netstat` to locate symbols in the kernel address space. As many of the commonly accessed symbols including routing tables, protocol control blocks etc. have been replaced by their virtualized counterparts residing in `struct vimage`, the `kldsym / kvm_nlist` programming interfaces were not able to automatically find the addresses of such symbols. Therefore, extensions to those functions were implemented in a similar manner as in the case of the `sysctl` interface, so that the virtualized symbols could be located using



the `kldsym` interface. Again, the virtual image instance to be searched in is determined based on `p_vimage` tag of the calling process.

In dynamically loadable kernel modules that implement functions closely related to network processing, such as *bridging* or *ipfw* packet filtering, the load / unload interfaces had to be updated. As multiple network stack instances might be active at the time of module loading, the module has to attach and initialize its private set of data in all of the network stack instances. During deactivation the reverse operation has to be performed, as all data structures in each network stack instance used by the module have to be freed.

### 4.3 Management interface

A basic API to implement management functions, such as creating, monitoring and modifying the properties of virtual images had to be introduced. The API uses a specialized `vi_req` structure to pass requests and return results to / from the kernel. As a minimum security precaution, regardless in which virtual image the process accessing the API is running, it has to have super-user (*root*) privileges to be allowed to perform any, even read-only operations.

**Creation and initialization of new virtual images** is the most basic function the API has to support, provided the current virtual image is allowed to create offspring. Similar to the standard UNIX processes, virtual images are logically organized in a hierarchical manner; despite being represented as only one linked list in the kernel. A *master* virtual image is always present, as it is created automatically upon system boot. A system running only with the *master* virtual image is practically indistinguishable from the standard FreeBSD OS. The *master* has the ultimate authority to manage the whole hierarchy of virtual images, and can empower its children to create more offspring, if desired.

To create a new virtual image, the kernel must accomplish several tasks. First it reserves memory to hold the new `struct vimage`, and inserts the new `struct` into the linked list of all virtual images. The kernel then creates a new loopback network instance and attaches it to the virtual image. Further, it calls initialization routines for all registered network protocols with a pointer to the new virtual image as argument, in a similar manner as during system startup in the original 4.4BSD networking code. Finally, special facilities such as *ipfw* firewalling are initialized for use in the new virtual image. During the whole sequence all interrupts are disabled, which is a sufficient locking method on a single-threaded kernel such as FreeBSD 4.8, however in the future a different

locking scheme has to be implemented in the process of migration to more recent OS versions with SMP support in the kernel.

At the time of creation, a *chroot* directory in which all the processes in the virtual image will run can be specified, together with a limit on the average percentage of CPU time the virtual image is allowed to consume. Although these parameters are optional, it is very likely they will be used in any serious virtual hosting scenario. The CPU percentage limit can be adjusted at any time; however changing the *chroot* directory requires no processes to exist within the virtual image. The *chrooted* directory tree has to be set up in the same manner as for the "classic" jailed environments.

**Deletion of virtual images** requires the protocol initialization sequences performed during the virtual image creation to be followed *backwards* during detachment of each network protocol instance. A virtual image can be deleted only when no user processes and sockets are attached to it. The deletion routine has to walk through all the configured protocol instances and gracefully free private memory structures used by each specific protocol suite, such as interface addresses, routing and hash tables, protocol control blocks etc. Furthermore, all the pending timers associated with protocols such as ARP or TCP have to be canceled, before the virtual image is unlinked from the global list of all virtual images, and the corresponding `struct vimage` can be freed.

**The network interfaces** can be *moved* from one virtual image to another. For example, this makes it possible for 802.1q VLAN interfaces bound to a "base" physical interface in one virtual image to be reassigned to other virtual images. Another use of this concept might be on machines with multiple physical interfaces to allow each interface to be assigned to an independent virtual image. As the interface moves from one virtual image to another it is automatically assigned a new index; for example if Ethernet interface `fxp1` is moved to another virtual image where no `fxp` interface instances exist, it will be renamed to `fxp0`.

**User processes** can use the API to switch to one of the children of their virtual image; however as a security precaution they cannot switch back to their parents or siblings. This in effect results for such switching to be used solely for spawning new processes in desired virtual images. As the only application to use this API, the `vimage` management utility has an option to prevent the new process from becoming *chrooted* before being spawned in the new virtual image, which can be useful for monitoring purposes and as a rescue option in



cases when the private directory tree in the target virtual image becomes damaged or unusable.

Although under normal circumstances it can be expected that management operations on virtual images will be executed relatively infrequently, the current implementation requires the whole kernel to be locked for the entire duration of all virtual image management system calls. Fortunately, the typical execution time of management functions is generally short enough not to introduce unacceptable delays, even on busy servers. Table 1 shows the typical execution duration of management operations, measured from the userland perspective on an otherwise idle system.

<i>Function</i>	<i>Duration</i>
Create a new vimage	212 $\mu$ s
Delete a vimage	72 $\mu$ s
Move network interface to a vimage	47 $\mu$ s
Switch a process to another vimage	20 $\mu$ s
Modify vimage parameters	21 $\mu$ s
Fetch statistics for one vimage	25 $\mu$ s

Table 1: Average execution time of virtual image management operations on an idle system

#### 4.4 Other pseudo-VM functions

Implementing a clonable network stack was the original and primary focus of the experimental code, without presumption whether the application will be virtual hosting, VPN provisioning, network simulation or something completely different. However, extending the experimental framework to provide some basic pseudo-VM functions, like hiding user processes running in one *virtualized* environment, turned out to be a fairly trivial task. This part of the implementation was easy because the basic framework offering such functionality was already there – the popular *jail* [2] facility in FreeBSD. The original *jail* implementation uses the `PRISON_CHECK(p1,p2)` macro at various points in the kernel to determine whether two processes are supposed to "see" and interact with each other by checking if they both belong to the same *jail*. It was sufficient to extend this macro by checking if the processes belong to the same network stack instance to achieve the same separation functionality originally present in *jails*. This basic facility was further extended with new functions related to management of CPU resources.

**CPU load and usage accounting** has been reorganized from system-wide to per virtual image, by migrating the `cp_time` and `averunnable` variables to

`struct vimage`. This allowed the CPU time spent in *user* and *system* contexts to be charged to the appropriate virtual image, which can help providing the administrators with better control and overview over the system behavior, and further restricting the global view on the system from within the virtual images.

The problem of accurately defining and finding the actual consumer of CPU time spent in the *interrupt* context is still an open area in OS research. Traditionally, UNIX systems pragmatically (and unfairly) charge the unlucky current process with the time spent in servicing interrupts. We found out that on the level of virtual images it is more practical to charge *all* virtual images for the time slice spent in the *interrupt* context, instead of only charging the current one. Such an approach simplified the implementation of individual per virtual image overall system load estimation. However, further experiments have shown that although it might be difficult or impossible to determine the actual consumer for a broad range of interrupt triggered events, for example for disk I/O DMA notifications, charging the appropriate virtual image for network traffic related interrupt context events can be implemented fairly simply and efficiently. Since the standard UNIX accounting is performed on statistical basis by periodically probing the state of CPU execution context (user, system, interrupt or idle), it was necessary to provide the accounting routine with more detailed information on which virtual image is currently active in the interrupt context. This was accomplished by introduction of a new global variable `vintr` which points to the current virtual image on which network processing is performed in the interrupt context.

As the interrupt context network processing in 4.4BSD kernels is split into two stages, the global `vintr` variable has to be set both on each entry to NICs device driver interrupt service routine and later during `netisr` processing. At exit the interrupt service routines are responsible to set the `vintr` back to `NULL`. When the `statclock()` accounting routine preempts a thread of execution running in the interrupt context, it checks whether `vintr` is set, in which case it can increment the corresponding counters only for the current virtual image. Otherwise, if `vintr` equals to `NULL`, *all* virtual images are charged for the consumed timeslice in interrupt context.

The described model of interrupt context time accounting is far from being accurate for two main reasons. First, it cannot properly classify the context switching and interrupt dispatching periods before the device driver can set the `vintr` variable, as well as after the `vintr` variable is cleared. Secondly, in the case when the traffic has to be *bridged* to virtual image

different from the one the physical NIC resides in, it is highly probable that over a longer period of time *both* virtual images will accrue interrupt context timeslots. Such an accounting is logically invalid, since only the virtual image that is the final destination of the incoming traffic should be charged. Nevertheless, despite not being entirely accurate, the per virtual image interrupt context accounting model can be a valuable indicator to system administrators in which virtual image to look for inbound traffic triggered CPU congestions.

**CPU usage limiting** was another function implemented as a straight follow up on the individual CPU usage accounting. Based on CPU usage limits set by the system administrator, the original 4.4BSD algorithm for scheduling the user processes on the CPU was extended to simply *skip* the processes in the active *run queue* belonging to virtual images that have recently consumed more CPU time than they were allowed to. As soon as it can be determined that the CPU usage limit has been exceeded, a process belonging to another virtual image is rescheduled, or the system enters the idle loop. A digital decay filter periodically lowers the accumulated per virtual image CPU usage averages, thus allowing the processes running in administratively constrained virtual images to be rescheduled later.

**Receive livelock** [14] is a situation when more incoming packets arrive than the system is capable to handle, resulting in the CPU being permanently locked into servicing interrupt requests. This presents a serious threat to stability of systems based on interrupt driven kernels, such as FreeBSD. In an OS split in multiple virtual machine environments the problem becomes even more emphasized since the entire system could become crippled under livelock resulting from only one network stack instance being flooded by excessive inbound traffic patterns (typical for today's frequent DoS attacks).

After successfully implementing per virtual image interrupt context CPU usage accounting, it was almost trivial to implement a simple feedback-based control algorithm for mitigating receive livelock. The interrupt context load is dampened by controllable discarding of received packets from the inbound protocol queues during *netisr()* processing. The interrupt context load threshold, based on which the decision is made whether to drop or pass the packets to further protocol processing, can be independently tuned for each virtual image. The described method is based on an assumption that in average more CPU cycles are consumed in protocol level processing than in device drivers, which only have to demultiplex received packets to the appropriate protocol inbound queue. Therefore

dropping the packets in the *netisr()* loop can significantly lower the CPU time spent in interrupt context processing only for traffic flows that require complex protocol-level processing (such as TCP or IPSEC), or in scenarios where complex packet filtering rules have to be traversed for each received packet. However, it should be noted that the above method for dampening interrupt context load does not provide an ultimate solution in mitigating receive livelocks. *Interrupt coalescing* and especially *polling* [15] can provide more adequate protection against livelocks under extreme overloads.

## 4.5 Memory resources

The kernel memory footprint is affected by the introduced modifications when running multiple network stacks. Compiled for the IA-32 platform, the modified kernel image file is 1663393 bytes big, which is an increase of only 5977 bytes compared to an equally configured unmodified kernel. However, the real issues with memory allocation arise when multiple virtual images are active in the system. Those issues can be classified in two groups.

**Various private memory pools** such as protocol control blocks (PCB), hash tables etc. which are created for each virtual image independently introduce specific implications. Although the size of *struct vimage* itself is relatively small (8012 bytes), during virtual image creation additional space for private memory pools has to be reserved. Thus *vmstat -m* reports that each virtual image consumes a total of approximately 23 Kbytes of kernel memory upon initialization. Unfortunately, this figure is not relevant, as it does not account for reserved but unused memory pages allocated by the *zone allocator* for private PCBs and the TCP syncache. Using the standard limits for maximum number of sockets and TCP syncache size, each virtual image could consume a total of 1161 pages or 4664 Kbytes of kernel memory. It is obvious that this mechanism presents a serious obstacle in scaling to large number of simultaneous virtual images. Optimal use of kernel memory could be accomplished by implementing common zone pools for all network stack instances; however, such an approach would violate the design requirement of making network stacks as independent as possible. An alternative model is therefore employed, which retains private memory pools, but enforces lower limits on number of sockets per virtual image, thus significantly reducing the per virtual image memory footprint. Such an approach offers additional tuning possibilities on resource limiting, since it allows system administrators to specify a per virtual image limit on the number of sockets.

The `mbuf` packet buffers present a different issue, as they are shared among all network stack instances. For example, a heavily utilized traffic shaper or delay queue in one network stack could bind and effectively exhaust all the `mbuf` buffers in the system, rendering other network stack instances unusable. Mitigation of these issues is not a trivial task to solve, as it would require implementing new limit checking mechanisms to the `mbuf` allocator, accompanied by additional checking done at all places in the kernel where `mbuf` buffers are either created or handed over from one network stack instance to another. As such an approach would require significant programming efforts it has not been implemented in the experimental network stack cloning framework. Therefore the system's stability depends on network stack instances to cooperatively share the global `mbuf` pools.

## 5 Performance

Extending any piece of software with a new functionality always rises the question what impact it will make on system performance. In the case of extending the network stack to support cloning, this question becomes highly emphasized since *all references* to symbols and variables used throughout the network stack have one additional level of indirection. To compare the performance between the standard and modified kernels, a series of simple tests have been run.

**Microbenchmarking tests** were performed in order to determine the execution duration of some typical kernel functions involved in packet processing. In each observed function, both in the standard and modified kernel, counter *start* and *stop* hooks were embedded for capturing the elapsed time. The CPU-embedded *TSC* clock cycle counter was used as the time reference. Execution of each observed section was provoked by appropriate external traffic on an otherwise completely idle test system.

Table 2 shows the comparative results of a series of

such tests. The first two columns mark the observed function and the test traffic used, followed by average execution duration and its standard deviation from the 5000 subsequent iterations, for the modified and standard kernel. Throughout the tests the modified kernel was running with only a single virtual image / network stack instance, resembling the functionality of the standard kernel.

From the broader series of tests, only those results are included that were both repeatable and with reasonably small standard deviation. Examining the results it becomes apparent that accurate measurement of execution time becomes more difficult as the observed code section becomes longer and more complex, which can be partially explained by the influence of phenomena such as system bus contentions, CPU cache coherence etc. Nevertheless, the test results are valuable in that they show no dramatic difference in execution time between code sections running on the modified and standard kernel. In fact, in the modified kernel some functions seem to execute slightly faster than in the standard one; however, as the functions shown in Table 2 present only a portion of tasks that the kernel performs in providing network communication, these small deviations can be considered insignificant.

**The effective throughput** for different types of traffic was measured during the second series of tests, again comparing the performance between standard and modified kernels. The test machine was powerful enough to easily drive the physical network interfaces at full media speed using most of the interesting traffic patterns, so it became apparent that testing with external traffic would not yield particularly usable results. Therefore it was decided to perform all the testing *inside* the referent machine, using it at the same time as source and drain of the test traffic. The traffic was looped back between traffic source and destination, eliminating the contentions that physical media such as Ethernet could introduce.

**The TCP throughput** was measured using the *netperf* [5] utility. Figure 5 shows the measured relation

<i>function</i>	<i>traffic</i>	clonable stack		standard stack		+/- cycles	+/- %
		<i>average</i>	<i>stdev</i>	<i>average</i>	<i>stdev</i>		
<code>ip_input()</code>	ICMP echo	224	9,5	144	12,6	80	56%
<code>ip_output()</code>	ICMP echo reply	502	29,3	492	17,3	10	2%
<code>icmp_input()</code>	ICMP echo	504	204,9	660	213,1	-156	-24%
<code>icmp_reflect()</code>	ICMP echo / reply	181	4,8	153	4,8	28	18%
<code>tcp_input()</code>	TCP data	4978	953,4	5263	1002,1	-285	-5%
<code>tcp_input()</code>	TCP ack	2280	479,1	2471	491,9	-191	-8%
<b>total</b>		<b>8669</b>		<b>9183</b>		<b>-514</b>	<b>-6%</b>

Table 2: Execution duration (in clock cycles) of certain packet processing functions in modified and standard kernel



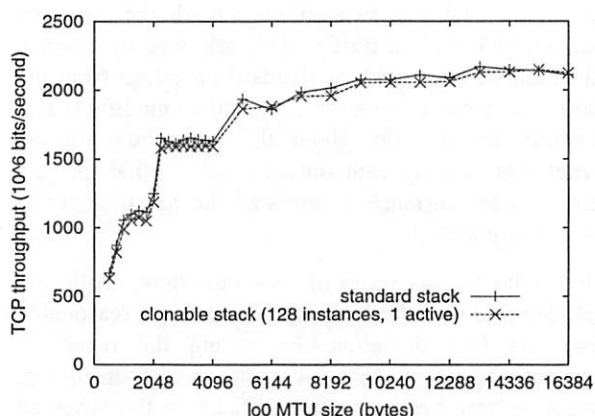


Figure 5: loopback TCP throughput

between maximum TCP throughput and the MTU size configured on the loopback network interface. It can be observed that the throughput obtained on the modified kernel running with a single network stack instance is marginally lower than on the unmodified kernel. For the MTU size of 1500 octets the achieved throughput using the modified kernel was around 93% of the value observed on the standard system. However, it should be noted that during this test the traffic passed through the network stack twice: once when data was transmitted by the server process (*netserver*), and once when the same data was received by the *netperf* client. The one-way throughput degradation is even less significant, and can be estimated as a square root of the obtained throughput ratio between standard and modified stack for both sending and receiving side processing. Therefore, for MTU=1500 we can estimate one-way maximum TCP throughput of the clonable network stack to be around 96.5% of the unmodified system.

**Measuring the maximum packet rate** for ICMP traffic, generated by the "flooding" `ping -qf` command and reflected by the kernel, yielded some interesting results. As shown in Figure 6, the maximum packet rate obtainable on the modified kernel was up to 5.7% higher than using the standard kernel. It is difficult to explain such a phenomenon, but it can be speculated that this might be due to better CPU cache coherency, since in the modified kernel most of the symbols involved in network processing are located close to each other in the *vimage* structure, while in the standard kernel they are interleaved with symbols used in other kernel functions not related to the network stack.

To summarize, all the presented measurements clearly indicate that the implemented network stack cloning code generally does not significantly degrade the system and network performance.

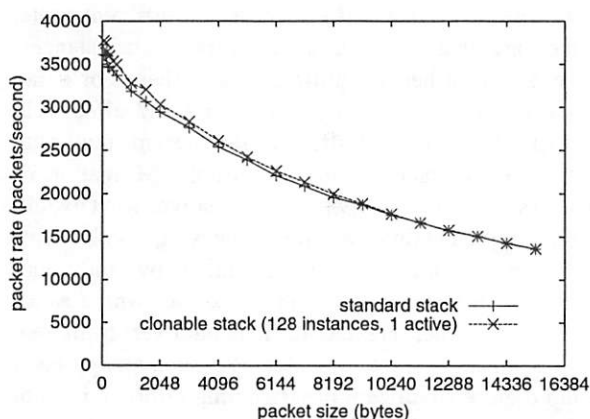


Figure 6: loopback packet rate

## 6 Application examples

The concept of network stack cloning was conceived with the goal of supporting a broad class of applications. Most system administrators will however be curious how the new framework fits in *virtual hosting scenarios*. In Figure 7 a simple virtual hosting configuration is shown where the system is split into three virtual images: "master", which is the default; and two subordinated virtual images called "client1" and "client2". The client virtual images are each assigned its private virtual Ethernet "ve" network interface, which are all bridged to the physical LAN through the real Ethernet interface (*fxp0*) residing in the "master" virtual image. The client virtual images reside in *chrooted* directory trees, which can be created using the standard methods for setting up the *jailed* environments, described in *jail* (8) online manual.

The following command sequence can be used to initially configure the described virtual hosting environment. The *vimage* command is used for

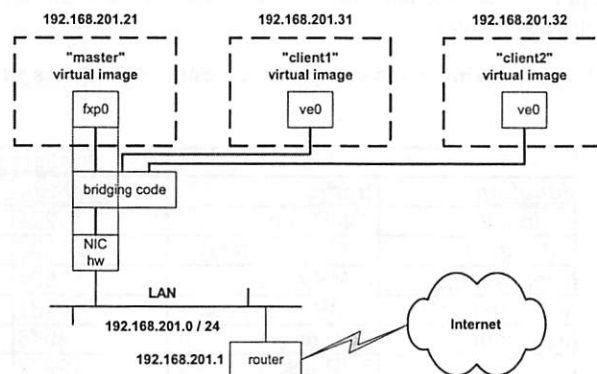


Figure 7: A simple virtual hosting scenario



managing the virtual images and network interfaces, with modifier determining the action to be performed. It is assumed the private directory tree for each client virtual image have been set up previously.

```
# create virtual ethernet interfaces
ifconfig ve0 create link 40:0:0:0:0:1
ifconfig ve1 create link 40:0:0:0:0:2

# create new virtual images
vimage -c client1 chroot /v/client1
vimage -c client2 chroot /v/client2

# move interfaces to new virtual images
vimage -i client1 ve0
vimage -i client2 ve1

# configure the bridge
sysctl \
    net.link.ether.bridge_cfg= \
    client1.ve0,client2.ve0,fxp0
sysctl net.link.ether.bridge=1
```

At this point no user processes exist in the "client" virtual images. Provided the virtual images are properly configured using the standard `rc.conf` file residing in their private directory tree, they could be started using the `/etc/rc` script, in similar manner as used for starting the standard *jails*. This could be accomplished using the following commands:

```
# start new virtual images
vimage client1 /bin/sh /etc/rc
vimage client2 /bin/sh /etc/rc
```

Unlike in jails, the parent virtual image can spawn a new process in its child at any time. The following example shows how IP configuration of virtual images can be performed manually:

```
vmbsd# vimage
master
vmbsd# vimage client1
Switched to vimage client1
# ifconfig
ve0: flags=8903
<UP,BROADCAST,PROMISC,SIMPLEX,MULTICAST>
mtu 1500 ether 40:00:00:00:00:01
lo0: flags=8008
<LOOPBACK,MULTICAST> mtu 16384
# ifconfig lo0 localhost
# ifconfig ve0 192.168.201.31
# route add default 192.168.201.1
# inetd
# ps -ax
  PID  TT  STAT      TIME COMMAND
   248  ??  Ss        0:00.02 inetd
   242  p1  S         0:00.06 vimage (csh)
   249  p1  R+        0:00.01 ps -ax
# hostname freenix
# exit
```

It is now possible to verify if the new virtual image can be accessed over the network:

```
vmbsd# telnet -K 192.168.201.31
Trying 192.168.201.31...
Connected to 192.168.201.31.
Escape character is '^]'.

FreeBSD/i386 (freenix) (ttty4)
...
```

The above example illustrates only the initial sequences in the management of virtual images. As at the first glance the achieved functionality might seem very similar to what can be done with the traditional FreeBSD jails, it should be noted that the real differences can be observed in the areas which are *not* supported in jails. Maintaining multiple IP addresses, independent packet filters and routing tables, access to the routing and raw sockets as well as `bpf` traffic capturing are among the key new functions the network stack cloning model introduces to the jail-styled virtual hosting scenarios.

The other applications that can benefit from the clonable network stack infrastructure range from fast and efficient real-time network simulations to advanced *overlayed* VPNs with independent and potentially overlapping addressing schemes. Unfortunately, the scope of this article limits further discussion on possible applications in these areas.

## 7 Conclusions and future work

The main contribution of this work is demonstrating the concept of network stack cloning can be efficiently implemented as an extension to the existing FreeBSD networking code. The experimental implementation has successfully preserved both the same overall performance level and full API/ABI compatibility with the original kernel, which were the two key requirements for adopting the network stack cloning model in a broad range of applications. When running with only a single network stack instance, looking from the userland perspective it is practically impossible to distinguish between a modified kernel and the original one, both regarding the general appearance, functionality, application interface, overall performance and memory footprint. However, by creating new network stack instances associated with *virtual images*, system administrators now can conveniently and efficiently partition the OS into highly independent *pseudo* or *light* virtual machine entities.

A substantial amount of work has yet to be completed before the network stack cloning model could be even considered for inclusion in an official FreeBSD source tree. As the experimental implementation covers only virtualization of basic IPv4 networking code, obviously the cloning support should be extended to other protocols, starting with IPv6 and IPSEC. Further, the framework should be ported and kept in sync with a development (*-current*) source tree, since the original experimental implementation is based on the *-stable* 4.x FreeBSD branch. And even if and when a highly polished patch against a development tree would become available, it is imminent that the question would arise whether the potential benefits of network stack cloning could outweigh the compatibility issues associated to maintaining different private or parallel source trees, which would become obsolete once the cloning patch would get integrated in the official kernel source tree.

In parallel with bringing the code in closer sync with the FreeBSD *-current* branch, the original concept of partitioning the OS in *virtual images* could be further extended by virtualizing other system resources, such as real and virtual memory, network and disk bandwidth, etc. An interesting option for further development could certainly be the reimplementing of *virtual images* as a modular *resource container* [6] type facility. Each resource instance (network stack, process group, CPU, memory etc.) would be represented by its own data structure, and `struct vimage` would only contain pointers to such structures. In such an environment, the system administrator could freely combine only the desired virtualized system resources in a *virtual image*, depending on the specific environment and application requirements.

The experimental code for network stack cloning support, along with additional technical information and application examples is available for download under a BSD-style license at <http://www.tel.fer.hr/zec/vimage/>. At the time of this writing, the source is maintained as a set of patches against the FreeBSD 4.8-RELEASE kernel.

## Acknowledgements

The author would like to thank Guido van Rooij, who was shepherding this work, for thorough and critical commentaries throughout the development of the paper, as well for all his previous encouragement and support in presenting the network stack cloning concept to the FreeBSD community. Many thanks go to anonymous reviewers for their valuable comments and suggestions.

Finally, the author would like to thank Julian Elischer for helping with presenting this work at the conference.

## References

- [1] Jeremy Sugerman, Ganesh Venkitachalam and Beng-Hong Lim: *Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor*, In Proc. of the USENIX Annual Technical Conference, 2001.
- [2] Poul-Henning Kamp and Robert Watson: *Jails: Confining the Omnipotent root*, In Proc. of the 2<sup>nd</sup> International SANE Conference, 2000.
- [3] Marshall K. McKusick et al.: *The design and implementation of the 4.4BSD operating system*, Addison-Wesley, 1996.
- [4] G. R. Wright and W. R. Stevens: *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, 1994.
- [5] netperf: <http://www.netperf.org/>
- [6] G. Banga, P. Druschel and J. C. Mogul: *Resource containers: A new facility for resource management in server systems*, In Proc. of the Symposium on Operating System Design and Implementation, 1999.
- [7] X. W. Huang, R. Sharma and S. Keshaw: *The ENTRAPID Protocol Development Environment*, In Proc. of the IEEE INFOCOM, 1999.
- [8] D. Ely, S. Savage and D. Wetherall: *Alpine: A User-Level Infrastructure for Network Protocol Development*, In Proc. of the 3<sup>rd</sup> USENIX Symposium on Internet Technologies and Systems, 2001.
- [9] S. Y. Wang and H. T. Kung: *A Simple Methodology for Constructing Extensible and High-Fidelity TCP/IP Network Simulators*, In Proc. of the IEEE INFOCOM, 1999.
- [10] Archie Cobbs: *All About Netgraph*, <http://ezine.daemonnews.org/200003/netgraph.html>
- [11] Riccardo Scandariato and Fulvio Rizzo: *Advanced VPN support on FreeBSD systems*, In Proc. of the 2<sup>nd</sup> European BSD Conference, 2002.
- [12] Luigi Rizzo: *Dummynet: A simple approach to the evaluation of network protocols*, ACM Computer Communication Review, 1997.
- [13] Jonathan Lemmon: *Resisting SYN flood DoS attacks with a SYN cache*, In Proc. of the BSDCon 2002.
- [14] J. C. Mogul and K. K. Ramakrishnan: *Eliminating receive livelock in an interrupt-driven kernel*, ACM Transactions on Computer Systems, Vol 15, No. 3, 1997.
- [15] Luigi Rizzo: *Device Polling support for FreeBSD*, <http://info.iet.unipi.it/~luigi/polling/>

# StarFish: highly-available block storage

Eran Gabber      Jeff Fellin      Michael Flaster      Fengrui Gu  
Bruce Hillyer      Wee Teck Ng      Banu Özden      Elizabeth Shriver

Information Sciences Research Center  
Lucent Technologies – Bell Laboratories  
600 Mountain Avenue, Murray Hill, NJ 07974  
{eran, jkf, mflaster, fgu, bruce, weeteck, ozden, shriver}@research.bell-labs.com

## Abstract

In this paper we present StarFish, a highly-available geographically-dispersed block storage system built from commodity servers running FreeBSD, which are connected by standard high-speed IP networking gear. StarFish achieves high availability by transparently replicating data over multiple storage sites. StarFish is accessed via a host-site appliance that masquerades as a host-attached storage device, hence it requires no special hardware or software in the host computer. We show that a StarFish system with 3 replicas and a write quorum size of 2 is a good choice, based on a formal analysis of data availability and reliability: 3 replicas with individual availability of 99%, a write quorum of 2, and read-only consistency gives better than 99.9999% data availability. Although StarFish increases the per-request latency relative to a direct-attached RAID, we show how to design a highly-available StarFish configuration that provides most of the performance of a direct-attached RAID on an I/O-intensive benchmark, even during the recovery of a failed replica. Moreover, the third replica may be connected by a link with long delays and limited bandwidth, which alleviates the necessity of dedicated communication links to all replicas.

## 1 Introduction

It is well understood that important data need to be protected from catastrophic site failures. High-end and mid-range storage systems, such as EMC SRDF [4] and Net-App SnapMirror [17], copy data to remote sites both to reduce the amount of data lost in a failure, and to decrease the time required to recover from a catastrophic site failure. Given the plummeting prices of disk drives and of high-speed networking infrastructure, we see the possibility of extending the availability and reliability advantages of on-the-fly replication beyond the realm of expensive, high-end storage systems. Moreover, we demonstrate advantages to having more than one remote replica of the data.

In this paper, we describe the StarFish sys-

tem, which provides host-transparent geographically-replicated block storage. The StarFish architecture consists of multiple replicas called storage elements (SEs), and a host element (HE) that enables a host to transparently access data stored in the SEs, as shown in Figure 1. StarFish is a software package for commodity servers running FreeBSD that communicate by TCP/IP over high-speed IP networks. There is no custom hardware needed to run StarFish. StarFish is mostly OS and machine independent, although it requires two device drivers (SCSI/FC target-mode driver and NVRAM driver) that we have implemented only for FreeBSD.

The StarFish project is named after the sea creature, since StarFish is designed to provide robust data recovery capabilities, which are reminiscent of the ability of a starfish to regenerate its rays after they are cut off.

StarFish is not a SAN (Storage Area Network), since SAN commonly refers to a Fibre Channel network with a limited geographical reach (a few tens of kilometers).

StarFish has several key achievements. First, we show that a StarFish system with the recommended configuration of 3 replicas (see Section 4) achieves good performance even when the third replica is connected by a communication line with a large delay and a limited bandwidth — a highly-available StarFish system does not require expensive dedicated communication lines to all replicas. Second, we show that StarFish achieves good performance during recovery from a replica failure, despite the heavy resource consumption of the data restoration activity. Generally, StarFish performance is close to that of a direct-attached RAID unit. Moreover, we present a general analysis that quantifies how the data availability and reliability depend on several system parameters (such as number of replicas, write quorum size, site failure rates, and site recovery speeds). This analysis leads to the suggestion that practical systems use 3 replicas and a write quorum size of 2.

In many real-world computing environments, a remote-replication storage system would be disqualified from consideration if it were to require special hardware

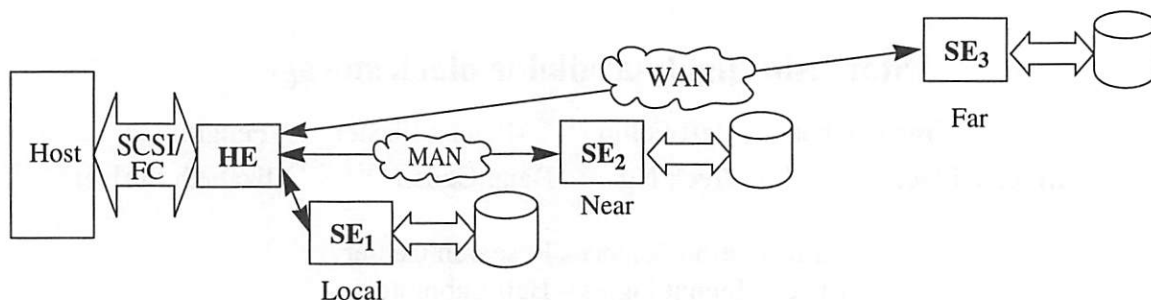


Figure 1: StarFish architecture and recommended setup.

in the host computer, or software changes in the operating system or applications. Replicating storage at the block level rather than at the file system level is general and transparent: it works with any host software and hardware that is able to access a hard disk. In particular, the host may use any local file system or a database that requires access to a hard disk, and not just a remote file system, such as NFS. The StarFish system design includes a host-site appliance that we call the host element (HE), which connects to a standard I/O bus on the host computer, as shown in Figure 1. The host computer detects the HE to be a pool of directly-attached disk drives; the HE transparently encapsulates all the replication and recovery mechanisms. In our prototype implementation, the StarFish HE connects to an Ultra-2 SCSI port (or alternately, to a Fibre Channel port) on the host computer.

If an application can use host-attached storage, it can equally well use StarFish. Thus, the StarFish architecture is broadly applicable — to centralized applications, to data servers or application servers on a SAN, and to servers that are accessed by numerous client machines in a multi-tier client/server architecture.

StarFish implements single-owner access semantics. In other words, only one host element can write to a particular logical volume. This host element may be connected to a single host or to a cluster of hosts by several SCSI buses. If we require the ability for several hosts to write to a single logical volume, this could be implemented by clustering software that prevents concurrent modifications from corrupting the data.

Large classes of data are owned (at least on a quasi-static basis) by a single server, for example in shared-nothing database architectures, and in centralized computing architectures, and for large web server clusters that partition the data over a pool of servers. The benefits that motivate the single-owner restriction are the clean serial I/O semantics combined with quick recovery/failover performance (since there is no need for distributed algorithms such as leader election, group membership, recovery of locks held by failed sites, etc.). By contrast, multiple-writer distributed replication incurs unavoidable tradeoffs among performance, strong

consistency, and high reliability as explained by Yu and Vahdat [25].

To protect against a site failure, a standby host and a standby HE should be placed in a different site, and they could commence processing within seconds using an up-to-date image of the data, provided that StarFish was configured with an appropriate number of replicas and a corresponding write quorum size. See Section 3 for details.

The remainder of this paper is organized as follows. Section 2 compares and contrasts StarFish with related distributed and replicated storage systems. Section 3 describes the StarFish architecture and its recommended configuration. Section 4 analyzes availability and reliability. Section 5 describes the StarFish implementation, and Section 6 contains performance measurements. We encountered several unexpected hurdles and dead ends during the development of StarFish, which are listed in Section 7. The paper concludes with a discussion of future work in Section 8 and concluding remarks in Section 9.

## 2 Related work

Many previous projects have established a broad base of knowledge on general techniques for distributed systems (e.g., ISIS [2]), and specific techniques applicable to distributed storage systems and distributed file systems. Our work is indebted to a great many of these; space limitations permit us to mention only a few.

The EMC SRDF software [4] is similar to StarFish in several respects. SRDF uses distribution to increase reliability, and it performs on-the-fly replication and updating of logical volumes from one EMC system to another, using synchronous remote writes to favor safety, or using asynchronous writes to favor performance. The first EMC system owns the data, and is the *primary* for the classic primary copy replication algorithm. By comparison, the StarFish *host* owns the data, and the HE implements primary copy replication to *multiple* SEs. StarFish typically uses synchronous updates to a subset of the SEs for safety, with asynchronous updates to additional SEs to increase availability. Note that this comparison is not



intended as a claim that StarFish has features, performance, or price equivalent to an EMC Symmetrix.

Petal [11] is a distributed storage system from Compaq SRC that addresses several problems, including scaling up and reliability. Petal's network-based servers pool their physical storage to form a set of virtual disks. Each block on a virtual disk is replicated on two Petal servers. The Petal servers maintain mappings and other state via distributed consensus protocols. By contrast, StarFish uses  $N$ -way replication rather than 2-way replication, and uses an all-or-none assignment of the blocks of a logical volume to SEs, rather than a declustering scheme. StarFish uses a single HE (on a quasi-static basis) to manage any particular logical volume, and thereby avoids distributed consensus.

Network Appliance SnapMirror [17] generates periodic snapshots of the data on the primary filer, and copies them asynchronously to a backup filer. This process maintains a slightly out-of-date snapshot on the backup filer. By contrast, StarFish copies all updates on-the-fly to all replicas.

The iSCSI draft protocol [19] is an IETF work that encapsulates SCSI I/O commands and data for transmission over TCP/IP. It enables a host to access a remote storage device as if it were local, but does not address replication, availability, and system scaling.

StarFish can provide replicated storage for a file system that is layered on top of it. This configuration is similar but not equivalent to a distributed file system (see surveys in [24] and [12]). By contrast with distributed file systems, StarFish is focused on replication for availability of data managed by a single host, rather than on unreplicated data that are shared across an arbitrarily scalable pool of servers. StarFish considers network disconnection to be a failure (handled by failover in the case of the HE, and recovery in case of an SE), rather than a normal operating condition.

Finally, Gibson and van Meter [6] give an interesting comparison of network-attached storage appliances, NASD, Petal, and iSCSI.

### 3 StarFish architecture

As explained in Section 1, the StarFish architecture consists of multiple storage elements (SEs), and a host element (HE). The HE enables the host to access the data stored on the SEs, in addition to providing storage virtualization and read caching. In general, one HE can serve multiple logical volumes and can have multiple connections to several hosts. The HE is a commodity server with an appropriate SCSI or FC controller that can function in target mode, which means that the controller can receive commands from the I/O bus. The HE has to run an appropriate target mode driver, as explained in Section 5.

We replicate the data in  $N$  SEs to achieve high availability and reliability; redundancy is a standard technique to build highly-available systems from unreliable components [21]. For good write performance, we use a quorum technique. In particular, the HE returns a success indication to the host after  $Q$  SEs have acknowledged the write, where  $Q$  is the *write quorum size*. In other words, StarFish performs synchronous updates to a quorum of  $Q$  SEs, with asynchronous updates to additional SEs for performance and availability.

Since StarFish implements single-owner access to the data, it can enforce consistency among the replicas by serialization: the HE assigns global sequence numbers to I/O requests, and the SEs perform the I/Os in this order to ensure data consistency. Moreover, to ensure that the highest update sequence number is up to date, the HE does not delay or coalesce write requests. To simplify failure recovery, each SE keeps the highest update sequence number (per logical volume) in NVRAM. This simple scheme has clear semantics and nice recovery properties, and our performance measurements in Section 6 indicate that it is fast.

Figure 1 shows a recommended StarFish setup with 3 SEs. The "local" StarFish replica is co-located with the host and the HE to provide low-latency storage. The second replica is "near" (e.g., connected by a dedicated high-speed, low-latency link in a metro area) to enable data to be available with high performance even during a failure and recovery of the local replica. A third replica is "far" from the host, to provide robustness in the face of a regional catastrophe. The availability of this arrangement is studied in Section 4, and the performance is examined in Section 6.

The HE and the SEs communicate via TCP/IP sockets. We chose TCP/IP over other reliable transmission protocols, such as SCTP [23], since TCP/IP stacks are widely available, optimized, robust, and amenable to hardware acceleration. Since many service providers sell Virtual Private Network (VPN) services, the HE may communicate with the far SE StarFish via a VPN, which provides communication security and ensures predictable latency and throughput. StarFish does not deal with communication security explicitly.

StarFish could be deployed in several configurations depending on its intended use. Figure 2 shows a deployment by an enterprise that has multiple sites. Note that in this configuration the local SE is co-located with the host and the HE. A storage service provider (SSP) may deploy StarFish in a configuration similar to Figure 1, in which all of the SEs are located in remote sites belonging to the SSP. In this configuration the local SE may not be co-located with the host, but it should be nearby for good performance.

StarFish is designed to protect against SE failures,

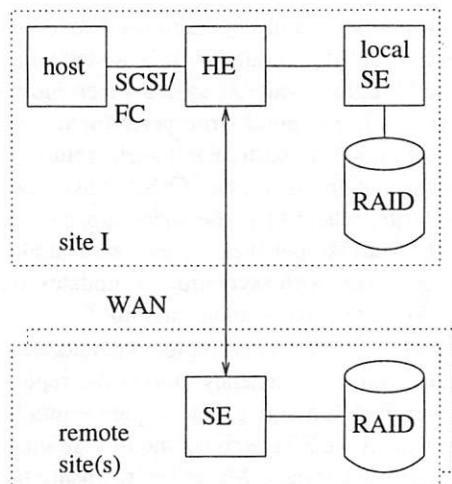


Figure 2: StarFish deployment in an enterprise.

network failure to some SEs, and HE failure. When an SE (or its RAID or network connection) fails, the HE continues to serve I/Os to the affected logical volumes, provided that  $Q$  copies are still in service. When the failed SE comes back up, it reconnects to the HE and reports the highest update sequence number of its logical volumes. This gives the HE complete information about what updates the SE missed. For each logical volume, the HE maintains a circular buffer of recent writes (the “write queue”). If an SE fails and recovers quickly (in seconds), it gets the missed writes from the HE. This recovery is called “quick recovery”. Also, each SE maintains a circular buffer of recent writes on a log disk. If an SE fails and recovers within a moderate amount of time (in hours — benchmark measurements from Section 6 suggest that the log disk may be written at the rate of several GB per hour) the HE tells the SE to retrieve the missed writes from the log disk of a peer SE. This recovery is called “replay recovery”. Finally, after a long failure, or upon connection of a new SE, the HE commands it to perform a whole-volume copy from a peer SE. This recovery is called “full recovery”. During recovery, the SE also receives current writes from the HE, and uses sequence number information to avoid overwriting these with old data. To retain consistency, the HE does not ask a recovering SE to service reads.

The fixed-size write queue in the HE serves a second purpose. When an old write is about to be evicted from this queue, the HE first checks that it has been acknowledged by all SEs. If not, the HE waits a short time (throttling), and if no acknowledgment comes, declares the SE that did not acknowledge as failed. The size of the write queue is an upper bound on the amount of data loss, because the HE will not accept new writes from the host until there is room in the write queue.

In normal operation (absent failures), congestion can-

not cause data loss. If any SE falls behind or any internal queue in the HE or the SE becomes full, the HE will stop accepting new SCSI I/O requests from the host.

The HE is a critical resource. We initially implemented a redundant host element using a SCSI switch, which would have provided automatic failure detection and transparent failover from the failed HE to a standby HE. However, our implementation encountered many subtle problems as explained in Section 7, so we decided to eliminate the redundant host element from the released code.

The current version of StarFish has a manually-triggered failover mechanism to switch to a standby HE when necessary. This mechanism sends an SNMP command to the SEs to connect to the standby HE. The standby HE can resume I/O activity within a few seconds, as explained in Section 6.4. The standby HE will assume that same Fibre Channel or SCSI ID of the failed HE. The manually-triggered failover mechanism can be replaced by an automatic failure detection and reconfiguration mechanism external to StarFish. One challenge in implementing such automatic mechanism is in the correct handling of network partitions and other communication errors. We can select one of the existing distributed algorithms for leader election for this purpose. If the HE connects to the host with Fibre Channel, the failover is transparent except for a timeout of the commands in transit. However, starting the standby HE on the same SCSI bus as the failed HE without a SCSI switch will cause a visible bus reset.

Since the HE acknowledges write completions only after it receives  $Q$  acknowledgments from the SEs, the standby HE can find the last acknowledged write request by receiving the highest update sequence number of each volume from  $Q$  SEs. If any SE missed some updates, the standby HE instructs it to recover from a peer SE.

## 4 Availability and reliability analysis

In this section we evaluate the availability and reliability of StarFish with respect to various system parameters. This enables us to make intelligent trade-offs between performance and availability in system design. Our main objectives are to quantify the availability of our design and to develop general guidelines for a highly available and reliable system.

The availability of StarFish depends on the SE failure ( $\lambda(t)$ ) and recovery ( $\mu(t)$ ) processes, the number of SEs ( $N$ ), the quorum size ( $Q$ ), and the permitted probability of data loss. The latter two parameters also bear on StarFish reliability. We assume that the failure and recovery processes of the network links and storage elements are independent identically distributed Poisson processes [3] with combined (i.e., network + SE) mean failure and recovery rates of  $\lambda$  and  $\mu$  failures and re-

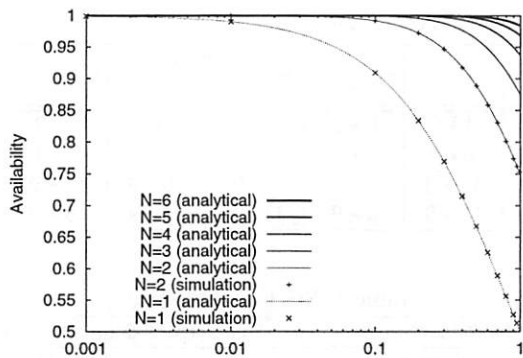


Figure 3: Availability with  $Q = 1$ ,  $N = 1-6$ .

coveries per second, respectively. Similarly, the HE has Poisson-distributed  $\lambda_{he}$  and  $\mu_{he}$ . Section 9 suggests ways to achieve independent failures in practice.

At time  $t$ , a component SE or HE is available if it is capable of serving data. We define the availability of StarFish as the steady-state probability that at least  $Q$  SEs are available. For example, a system that experiences an 1-day outage every 4 months is 99% available. We assume that if the primary HE fails, we can always reach the available SEs via a backup HE.

We define the reliability of a system as the probability of no data loss. For example, a system that is 99.9% reliable has a 0.1% probability of data loss.

#### 4.1 Availability results

The steady-state availability of StarFish,  $A(Q, N)$ , is derived from the standard machine repairman model [3] with the addition of a quorum. It is the steady-state probability that at most  $Q$  SEs are down and at least  $(N - Q)$  SEs are up. Thus, the steady-state availability can be expressed as

$$A(Q, N) = \frac{\sum_{i=0}^{N-Q} \binom{N}{i} \rho^i}{(1 + \rho)^N} \quad (1)$$

where  $\rho = \lambda/\mu$  is called the load, and  $1 \leq Q \leq N \forall Q, N$ . Eq. 1 is valid for  $0 < \rho < 1$  (i.e., when the failure rate is less than the recovery rate). Typical values of  $\rho$  range from 0.1 to 0.001, which correspond to hardware availability of 90% to 99.9%. See [5] for derivation of Eq. 1.

Figure 3 shows the availability of StarFish using Eq. 1 with a quorum size of 1 and increasing number of SEs. We validate the analytical model up to  $N = 3$  with an event-driven simulation written using the `smpl` simulation library [13]. Because the analytical results are in close agreement with the simulation results, we will not present our simulation results in rest of this paper.

We observe from Figure 3 that availability increases with  $N$ . This can also be seen from Eq. 1, which is strictly monotonic and converges to 1 as  $N \rightarrow \infty$ . We

also note that availability  $\rightarrow 1$  as  $\rho \rightarrow 0$ . This is because the system becomes highly available when each SE seldom fails or recovers quickly.

We now examine the system availability for *typical* configurations. Table 1 shows StarFish's availability in comparison with a single SE. We use a concise availability metric widely used in the industry, which counts the number of 9s in an availability measure. For example, a system that is 99.9% available is said to have three 9s, which we denote  $3 \star 9$ . We use a standard design technique to build a highly available system out of redundant unreliable components [21]. The SE is built from commodity components and its availability ranges from 90% [14] to 99.9% [7]. StarFish combines SEs to achieve a much higher system availability when  $N = 2Q + 1$ . For example, Table 1 indicates that if the SEs are at least 99% available, StarFish with a quorum size of 1 and 3 SEs is 99.9999% available ( $6 \star 9$ ).

We also notice from Table 1 that, for fixed  $N$ , StarFish's availability *decreases* with larger quorum size. In fact, for  $Q = N = 3$ , StarFish is less available than a single SE, because the probability of keeping all SEs concurrently available is lower than the probability that a single one is available. Increasing quorum size trades off availability for reliability. We quantify this trade-off next.

#### 4.2 Reliability results

StarFish can potentially lose data under certain failure scenarios. Specifically, when  $Q \leq \lfloor N/2 \rfloor$ , StarFish can lose data if the HE and  $Q$  SEs containing up-to-date data fail. The *amount* of data loss is bounded by the HE write queue size as defined in Section 3. This queue-limited exposure to data loss is similar to the notion of time-limited exposure to data loss widely used in file systems [20]. We make this trade-off to achieve higher performance and availability at the expense of a slight chance of data loss. The *probability* of data loss is bounded by  $1/4^Q$ , which occurs when  $\rho = 1$  and  $\rho_{he} = 0$ . This implies that the lowest reliability occurs when  $Q = 1$ , and the reliability increases with larger  $Q$ .

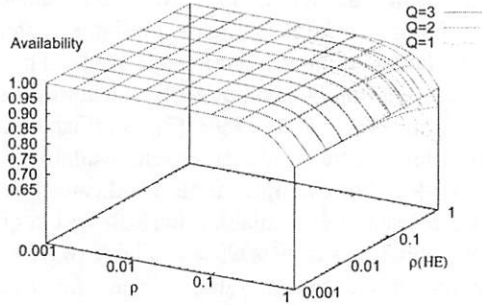
We note that there is no possibility of data loss if  $Q > \lfloor N/2 \rfloor$  and at least  $Q$  SEs are available. This is because when we have a quorum size which requires the majority of SEs to have up-to-date data when available, failures in the remaining SEs do not affect system reliability as we still have up-to-date data in the  $Q$  remaining SEs. However, this approach can reduce availability (see Table 1) and performance (see Section 6.3).

Another approach is to trade-off the system functionality while still maintaining the performance and reliability requirements. For example, we may allow StarFish to be available in a *read-only* mode during failure, which we call StarFish with *read-only consistency*.



Table 1: Availability of StarFish for typical configurations.

SE availability	$Q = 1$			$Q = 2$			$Q = 3$			
	$N = 2$	$N = 3$	$N = 4$	$N = 3$	$N = 4$	$N = 5$	$N = 3$	$N = 5$	$N = 6$	$N = 7$
90.00% (1 * 9)	2 * 9	3 * 9	4 * 9	1 * 9	2 * 9	3 * 9	0 * 9	2 * 9	3 * 9	3 * 9
99.00% (2 * 9)	4 * 9	6 * 9	8 * 9	3 * 9	5 * 9	7 * 9	1 * 9	5 * 9	6 * 9	8 * 9
99.90% (3 * 9)	6 * 9	9 * 9	12 * 9	5 * 9	8 * 9	11 * 9	2 * 9	8 * 9	10 * 9	13 * 9
99.99% (4 * 9)	8 * 9	12 * 9	16 * 9	7 * 9	11 * 9	15 * 9	3 * 9	11 * 9	14 * 9	18 * 9

Figure 4: Availability with  $N = 3$ ,  $Q = 1-3$ , read-only consistency.

Read-only mode obviates the need for  $Q$  SEs to be available to handle updates. This increases the availability of the system, as it is available for reads when the HE and at least 1 SE is available, or if any SE with up-to-date data is available and we fail over to a standby HE as described in Section 3. With read-only consistency, StarFish has steady-state availability  $A_{\text{ReadOnly}}(Q, N)$  of

$$\frac{\sum_{i=0}^{N-1} \binom{N}{i} \rho^i}{(1 + \rho_{he})(1 + \rho)^N} + \frac{\rho_{he} \sum_{i=0}^{Q-1} \binom{Q}{i} \rho^i}{(1 + \rho_{he})(1 + \rho)^Q}. \quad (2)$$

Eq. 2 is derived using Bayes Theorem [3]. We assume that the HE and SEs fail independently. The availability of StarFish with read-only consistency is union of two disjoint events: the HE and any  $Q$  SEs are alive, the HE fails and  $Q$  distinct SEs with current updates are alive.

Figure 4 shows the availability of StarFish with read-only consistency with respect to the SE load,  $\rho$ , and HE load (i.e.,  $\rho(HE)$  in Figure 4). We observe that when the HE is always available (i.e.,  $\rho(HE) = 0$ ), StarFish's availability is independent of the quorum size since it can always recover from the HE. When the HE becomes less available, we observe that StarFish's availability *increases* with larger quorum size. Moreover, the largest increase occurs from  $Q = 1$  to  $Q = 2$  and is bounded by  $3/16$  when  $\rho = 1$ . This implies a diminishing gain in availability beyond  $Q = 2$ , and we recommend using a quorum size of 2 for StarFish with read-only consistency. Table 2 shows StarFish's availability with read-only consistency for typical failure and recovery rates.

We suggest that practical systems select  $Q = 2$ , since a larger  $Q$  offers diminishing improvements in availabil-

Table 3: StarFish code size

component	language	source lines
host element (HE)	C++	9,400
storage element (SE)	C++	8,700
common code	C/C++	18,000
SCSI target mode driver	C	5,700
NVRAM driver	C	700
total		42,500

ity. To prevent data loss,  $N$  must be  $< 2Q$ . Thus we suggest that  $N = 3$  and  $Q = 2$  is a reasonable choice. In this configuration, StarFish with SE availability of 99% and read-only consistency is 99.9999% available ( $6 * 9$ ).

Increasing  $N$  in general reduces the performance, since it increases the load on the HE. The only way that a large  $N$  can improve performance is by having more than one local SE. In this configuration the local SEs can divide the load of read requests without incurring transmission delays. Alternately, all local SE may receive the same set of read requests, and the response from the first SE is sent to the host.

Although StarFish may lose data with  $Q = 1$  after a single failure, the  $Q = 1$  configuration may be useful for applications requiring read-only consistency, and applications that can tolerate data inconsistency like news-group and content distribution. Note that  $Q = 1$  is the current operating state for commercial systems that are not using synchronous remote copy mechanisms, so this configuration is useful as a reference.

## 5 Implementation

Table 3 shows the number of source lines and language for various components of StarFish. The "common code" includes library functions and objects that are used by both the HE and SE. The HE and SE are user-level processes, which provide better portability and simplify debugging relative to kernel-level servers. The main disadvantage of user-level processes is extra data copies to/from user buffers, which is avoided by kernel-level servers. Since the main bottleneck of the HE is the TCP/IP processing overhead (see Section 9), this is a reasonable tradeoff.

The HE and SE are multi-threaded. They use the LinuxThreads package, which is largely compatible with



Table 2: Availability of StarFish with read-only consistency for typical failure and recovery rates.

SE availability	$Q = 1$			$Q = 2$			$Q = 3$		
	$N = 2$	$N = 3$	$N = 4$	$N = 3$	$N = 4$	$N = 5$	$N = 3$	$N = 4$	$N = 5$
90.00% (1 * 9)	2 * 9	3 * 9	4 * 9	3 * 9	4 * 9	5 * 9	3 * 9	4 * 9	5 * 9
99.00% (2 * 9)	4 * 9	5 * 9	6 * 9	6 * 9	7 * 9	8 * 9	6 * 9	8 * 9	9 * 9
99.90% (3 * 9)	5 * 9	6 * 9	7 * 9	8 * 9	9 * 9	10 * 9	9 * 9	11 * 9	12 * 9
99.99% (4 * 9)	7 * 9	8 * 9	8 * 9	11 * 9	12 * 9	12 * 9	12 * 9	15 * 9	16 * 9

Pthreads [16]. LinuxThreads creates a separate process for each thread with the `rfork()` system call, enabling the child process to share memory with its parent.

StarFish maintains a collection of log files. There is a separate log file per StarFish component (HE or SE). StarFish closes all active log files once a day and opens new ones, so that long-running components have a sequence of log files, one per day. We found that StarFish initially suffered from many bugs and transient problems that caused failure and recovery of one component without causing a global failure. Scanning the log files once a day for failure and recovery messages was invaluable for detecting and correcting these problems.

StarFish code expects and handles failures of every operation. Every routine and method returns a success or failure indication. The calling code has the opportunity to recover or propagate the failure. The following code excerpt illustrates StarFish's error handling.

```
if ((code = operation(args)) != OK) {
    // log the failure
    user_report(severity,
        "operation failed due to %s",
        get_error_text(code));
    // propagate the error
    return code;
}
```

There are several reasons for choosing this coding style over exceptions. First and foremost, this coding style is applicable for C and C++. Second, it encourages precise handling of failures at the place they occurred, instead of propagating them to an exception handler in a routine higher in the calling chain. Third, failures are the rule rather than the exception in a distributed system. Handling failures close to the place they occur enables StarFish to recover gracefully from many transient failures.

StarFish has two new device drivers: a SCSI target mode driver, and an NVRAM driver. These drivers are the only part of the project that is operating-system dependent. The drivers are written in C for FreeBSD versions 4.x and 5.x.

The SCSI target mode driver enables a user-level server to receive incoming SCSI requests from the SCSI bus and to send responses. In this way, the server emulates the operation of a SCSI disk. The target driver uses

the FreeBSD CAM [8] subsystem to access Adaptec and Qlogic SCSI controllers. It can also be used to access any other target mode controllers that have a CAM interface, such as Qlogic Fibre Channel controllers.

As a side note, we could not use Justin Gibbs' target mode driver since it was not sufficient for our needs when we started the project. By the time Nate Lawson's target mode driver [10] was available, our driver implementation was completed. In addition, our driver supports tagged queuing, which does not appear to be supported by Lawson's driver.

The NVRAM driver maps the memory of Micro Memory MM5415 and MM5420 non-volatile memory (NVRAM) PCI cards to the user's address space. The HE and the SE store persistent information that is updated frequently in the NVRAM, such as update sequence numbers.

## 6 Performance measurements

This section describes the performance of StarFish for representative network configurations and compares it to a direct-attached RAID unit. We present the performance of the system under normal operation (when all SEs are active), and during recoveries. We also investigate the performance implications of changing the parameters of the recommended StarFish configuration.

### 6.1 Network configurations, workloads, and testbed

We measure the performance of StarFish in a configuration of 3 SEs (local, near, and far), under two emulated network configurations: a dark-fiber network, and a combination of Internet and dark fiber. In our testbed, the dark-fiber links are modeled by gigabit Ethernet (GbE) with the FreeBSD `dummynet` [18] package to add fixed propagation delays. The Internet links are also modeled by GbE, with `dummynet` applying both delays and bandwidth limitations. We have not studied the effects of packet losses, since we assume that a typical StarFish configuration would use a VPN to connect to the far SE for security and guaranteed performance. This is also the reason we have not studied the effects of Internet transient behavior.

Since the speed of light in optical fiber is approximately 200km per millisecond, a one-way delay of 1ms

Table 4: PostMark parameters

parameter	value
# files	40904
# transactions	204520
median working set size (MB)	256
host VM cache size (MB)	64
HE cache size (MB)	128

represents the distance between New York City and a back office in New Jersey, and a delay of 8ms represents the distance between New York City and Saint Louis, Missouri. The delay values for the far SE when it is connected by Internet rather than dark fiber (values from the AT&T backbone [1]) are 23ms (New York to Saint Louis), 36ms (continental US average), and 65ms (New York to Los Angeles). We use Internet link bandwidths that are 20%, 33%, 40%, 60%, and 80% of an OC-3 line, i.e., 31Mb/s, 51Mb/s, 62Mb/s, 93Mb/s, and 124Mb/s.

We measure the performance of StarFish with a set of micro-benchmarks (see Section 6.3) and PostMark version 1.5. PostMark [9] is a single-threaded synthetic benchmark that models the I/O workload seen by a large email server. The email files are stored in a UNIX file system with soft updates disabled. (Disabling soft updates is a conservative choice, since soft updates increase concurrency, thereby masking the latency of write operations.) We used the PostMark parameters depicted in Table 4. Initial file size is uniformly distributed between 500 and 10,000 bytes, and files never grow larger than 10,000 bytes. In all of the following experiments, we measured only performance of the transactions phase of the PostMark benchmark.

In FreeBSD, the VM cache holds clean file system data pages, whereas the buffer cache holds dirty data [15, section 4.3.1]. The host VM cache size and the HE cache size are chosen to provide 25% and 50% read hit probability, respectively. Because the workload is larger than the host VM cache, it generates a mixture of physical reads and writes. We control the VM cache size by changing the kernel's physical memory size prior to system reboot. We verified the size of the host VM cache for every set of parameters to account for the memory that is taken by the OS kernel and other system processes that are running during the measurements.

Figure 5 depicts our testbed configuration of a single host computer connected to either (a) a direct-attached RAID unit, or (b) a StarFish system with 3 SEs. All RAID units in all configurations are RaidWeb Arena II with 128MB write-back cache and eight IBM Deskstar 75GXP 75GB 7,200 RPM EIDE disks and an external Ultra-2 SCSI connection. The HE and SEs are connected via an Alteon 180e gigabit Ethernet switch. The host computer running benchmarks is a Dell Pow-

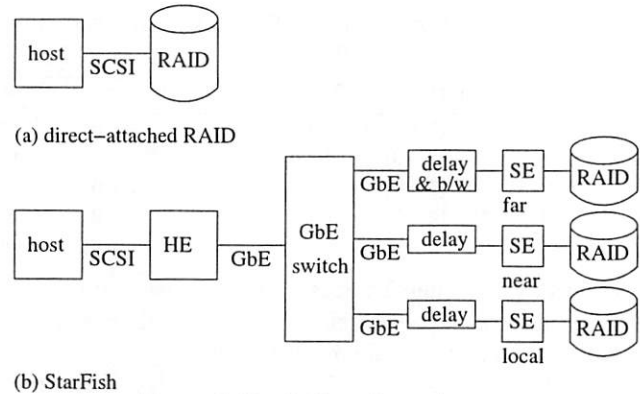


Figure 5: Testbed configuration.

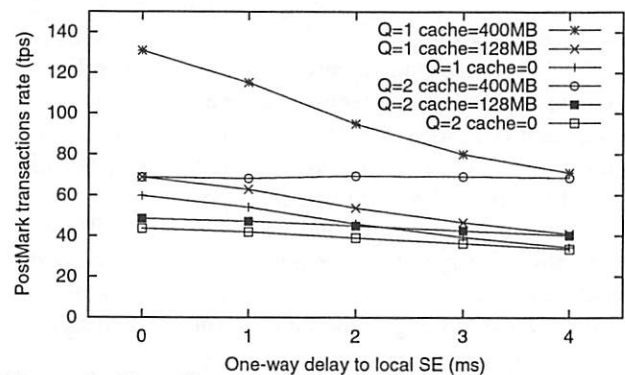


Figure 6: The effect of the one-way delay to the local SE and the HE cache size on PostMark transaction rate. The near SE and the far SE have one-way delays of 4 and 8ms, respectively.

erEdge 2450 server with dual 733 MHz Pentium III processors running FreeBSD 4.5. The HE is a Super-Micro 6040 server with dual 1GHz Pentium III processors running FreeBSD 4.4. The SEs are Dell PowerEdge 2450 servers with dual 866 MHz Pentium III processors running FreeBSD 4.3.

## 6.2 Effects of network delays and HE cache size on performance

In this section we investigate variations from the recommended setup to see the effect of delay to the local SE and the effect of the HE cache. Figure 6 shows the effects of placing the local SE farther away from the HE, and the effects of changing the HE cache size. In this graph the near SE and the far SE have one-way delays of 4 and 8ms, respectively. If the HE cache size is 400MB, all read requests hit the HE cache for this workload. The results of Figure 6 are not surprising. A larger cache improves PostMark performance, since the HE can respond to more read requests without communicating with any SE. A large cache is especially beneficial when the local SE has significant delays. The benefits of using a

cache to hide network latency have been established before (see, for instance, [15]).

However, a larger cache does not change the response time of write requests, since write requests must receive responses from  $Q$  SEs and not from the cache. This is the reason PostMark performance drops with increasing latency to the local SE for  $Q = 1$ . When  $Q = 2$ , the limiting delay for writes is caused by the near SE, rather than the local SE. The performance of  $Q = 2$  also depends on the read latency, which is a function of the cache hit rate and the delay to the local SE for cache misses.

All configurations in Figure 6 but one show decreasing transaction rate with increasing delay to the local SE. The performance of the configuration  $Q = 2$  and 400MB cache size is not influenced by the delay to the local SE, because read requests are served by the cache, and write requests are completed only after both the local and the near SE acknowledge them.

In summary, the performance measurements in Figure 6 indicate that with  $Q = 1$  StarFish needs the local SE to be co-located with the HE; with  $Q = 2$  StarFish needs a low delay to the near SE; and the HE cache significantly improves performance.

### 6.3 Normal operation and placement of the far SE

To examine the performance of StarFish during normal operation, we use micro-benchmarks to reveal details of StarFish's performance, and PostMark to show performance under a realistic workload.

The 3 micro-benchmarks are as follows. **Read hit:** after the HE cache has been warmed, the host sends 50,000 random 8KB reads to a 100MB range of disk addresses. All reads hit the HE cache (in the StarFish configuration), and hit the RAID cache (in the host-attached RAID measurements). **Read miss:** the host sends 10,000 random 8KB reads to a 2.5GB range of disk addresses. The HE's cache is disabled to ensure no HE cache hits. **Write:** the host sends 10,000 random 8KB writes to a 2.5GB range of disk addresses.

We run a variety of network configurations. There are 10 different dark-fiber network configurations: every combination of one-way delay to the near SE that is one of 1, 2, or 4ms, and one-way delay to the far SE that is one of 4, 8, or 12ms. The 10th configuration has one-way delay of 8ms to both the near and far SEs. In all configurations, the one-way delay to the near SE is 0. We also present the measurements of the Internet configurations which are every combination of one-way delay to the far SE that is one of 23, 36, or 65ms, and bandwidth limit to the far SE that is one of 31, 51, 62, 93, or 124Mbps. In all of the Internet configurations the one-way delay to the local and near SEs are 0 and 1ms,

Table 5: StarFish average write latency on dark-fiber and Internet network configurations with  $N=3$ . The local SE has a one-way delay of 0 in all configurations. The Internet configurations have one-way delay to the far SE that ranges from 23 to 65 ms and bandwidth limit that ranges from 31 to 124 Mb/s. Maximum difference from the average is 5%.

	near SE delay	configuration far SE delay/bandwidth	# conf.	write latency (ms)
RAID	-	-	1	2.6
dark fiber configurations:				
$Q = 1$	1-8	4-12	10	2.6
$Q = 2$	1	4-12	3	3.4
$Q = 2$	2	4-12	3	5.3
$Q = 2$	4	4-12	3	9.2
$Q = 2$	8	8	1	17.2
Internet configurations:				
$Q = 1$	1	23-65/31-124	4	2.6
$Q = 2$	1	23-65/31-124	4	3.4

Table 6: StarFish micro-benchmarks on dark-fiber networks with  $N=3$ . Maximum difference from the average is 7%. The results of the Internet network configurations are equivalent to the corresponding dark-fiber network configurations.

config- uration	# conf.	single-threaded latency (ms)		multi-threaded throughput (MB/s)		
		read miss	read hit	write	read miss	read hit
RAID	1	7.2	0.4	3.0	4.9	25.6
$Q = 1$	10	9.4	0.4	3.0	4.6	26.3
$Q = 2$	10	9.4	0.4	3.0	4.6	26.3

respectively.

Table 5 shows the write latency of the single-threaded micro-benchmark on dark-fiber network configurations. The write latency with  $Q = 1$  is independent of the network configuration, since the acknowledgment from the local SE is sufficient to acknowledge the entire operation to the host. However, the write latency with  $Q = 2$  is dependent on the delay to the near SE, which is needed to acknowledge the operation. For every millisecond of additional one-way delay to the near SE, the write latency with  $Q = 2$  increases by about 2ms, which is the added round-trip time. The write latency on the Internet configurations is the same as the corresponding dark-fiber configurations, since the local SE (for  $Q = 1$ ) and near SE (for  $Q = 2$ ) have the same delay as in the dark-fiber configuration.

Table 6 shows the micro-benchmark results on the same 10 dark-fiber network configurations as in Table 5. Table 6 shows that the read miss latency is independent of the delays and bandwidth to the non-local SEs be-



Table 7: Average PostMark performance on dark-fiber and Internet networks with 2 and 3 SEs. The local SE has a one-way delay of 0 in all configurations. The Internet configurations have one-way delay to the far SE that is either 23 or 65ms and bandwidth limit that is either of 51 or 124Mbps. Maximum difference from the average is 2%. For this workload, PostMark performs 85% writes.

Q	near SE delay	configuration		# conf.	Post- Mark (tps)
		far SE delay/band.			
RAID				1	73.12
dark fiber configurations $N=2$ :					
1	1	-	1	71.01	
2	1	-	1	65.64	
dark fiber configurations $N=3$ :					
1	1-8	4-12	10	68.80	
2	1	4-12	3	63.85	
2	2	4-12	3	57.97	
2	4	4-12	3	48.57	
2	8	8	1	35.53	
Internet configurations $N=3$ :					
1	1	{23,65}/{51,124}	4	67.98	
2	1	{23,65}/{51,124}	4	62.46	

cause StarFish reads from the closest SE. The read hit latency is fixed in all dark-fiber configurations, since read hits are always handled by the HE. The latency of the Internet configurations is equivalent to the dark-fiber configurations since no read requests are sent to the far SE. Tables 5 and 6 indicate that as long as there is an SE close to the host, StarFish's latency with  $Q = 1$  nearly equals a direct-attached RAID.

To examine the throughput of concurrent workloads, we used 8 threads in our micro-benchmarks, as seen in Table 6. StarFish throughput is constant across all ten dark-fiber network configurations and both write quorum sizes. The reason is that read requests are handled by either the HE (read hits) or the local SE (read misses) regardless of the write quorum size. Write throughput is the same for both write quorum sizes since it is determined by the throughput of the slowest SE and not by the latency of individual write requests. For all tested workloads, StarFish throughput is within 7% of the performance of a direct-attached RAID.

It is important to note that Starfish resides between the host and the RAID. Although there are no significant performance penalties in the tests described above, the HE imposes an upper bound on the throughput of the system because it copies data multiple times. This upper bound is close to 25.7MB, which is the throughput of reading data from the SE cache by 8 concurrent threads with the HE cache turned off. The throughput of a direct-attached RAID for the same workload is 52.7MB/s.

Table 7 shows that PostMark performance is influenced mostly by two parameters: the write quorum size, and the delay to the SE that completes the write quorum. In all cases the local SE has a delay of 0, and it responds to all read requests. The lowest bandwidth limit to the far SE is 51Mbps and not 31Mbps as in the previous tests, since PostMark I/O is severely bounded at 31Mbps. When  $Q = 1$ , the local SE responds first and completes the write quorum. This is why the performance of all network configurations with  $N = 3$  and  $Q = 1$  is essentially the same. When  $Q = 2$ , the response of the near SE completes the write quorum. This is why the performance of all network configurations with  $N = 3$  and  $Q = 2$  and with the same delay to the near SE is essentially the same.

An important observation of Table 7 is that there is no performance reason to prefer  $N = 2$  over  $N = 3$ , and  $N = 3$  provides higher availability than  $N = 2$ . As expected, the performance of  $N = 2$  is better than the corresponding  $N = 3$  configuration. However, the performance of  $N = 3$  and  $Q = 2$  with a high delay and limited bandwidth to the far SE is at least 85% of the performance of a direct-attached RAID. Another important observation is that StarFish can provide adequate performance when one of the SEs is placed in a remote location, without the need for a dedicated dark-fiber connection to that location.

## 6.4 Recoveries

As explained in Section 3, StarFish implements a manual failover. We measured the failover time by running a script that kills the HE process, and then sends SNMP messages to 3 SEs to tell them to reconnect to a backup HE process (on a different machine). The SEs report their current sequence numbers for all the logical volumes to the backup HE. This HE initiates replay recoveries to bring all logical volumes up to date. With  $N = 3$ ,  $Q = 2$ , and one logical volume, the elapsed time to complete the failover ranges from 2.1 to 2.2 seconds (4 trials), and the elapsed time with four logical volumes ranges from 2.1 to 3.2 seconds (9 trials), which varies with the amount of data that needs to be recovered.

Table 8 shows the performance of the PostMark benchmark in a system where one SE is continuously recovering. There is no idle time between successive recoveries. The replay recovery recovers the last 100,000 write requests, and the full recovery copies a 3GB logical volume. Table 8 shows that the PostMark transaction rate during recovery is within 74–98% of the performance of an equivalent system that is not in recovery, or 67–90% of the performance of a direct-attached RAID.

The duration of a recovery is dependent on the the recovery type (full or replay). Table 8 shows that on a dark-fiber network, replay recovery transfers 7.53–



Table 8: Average PostMark performance during recovery. The one-way delay to the local and near SEs in all configurations are 0 and 1 ms, respectively. The Internet configurations have one-way delay to the far SE that is either 23 or 65ms, and bandwidth limit that is either 51 or 124Mbps. Maximum difference from the average transaction rate is 7.3%. PostMark transactions rate of a direct-attached RAID is 73.12 tps.

Q	far SE delay/bandwidth	recovering SE	# conf.	replay recovery		full recovery		no recovery PostMark (tps)
				PostMark (tps)	recovery rate (MB/s)	PostMark (tps)	recovery rate (MB/s)	
1	8	local,near,far	3	60.65	8.28–9.89	53.24	7.99–10.12	68.80
1	{23,65}/{51,124}	far	4	63.92	2.52–5.62	60.09	2.97–6.32	67.98
2	8	local,near,far	3	58.41	7.53–9.02	49.64	8.64–9.22	63.85
2	{23,65}/{51,124}	far	4	59.92	2.53–5.79	56.85	2.68–6.64	62.46

9.89MB/s. If the far SE recovers over a 51Mbps Internet link, the average replay recovery rate is about 2.7MB/s. When the Internet bandwidth is 124Mbps and one-way delay is 65ms, our TCP window size of 512KB becomes the limiting factor for recovery speed, because it limits the available bandwidth to  $window\_size/round\_trip\ time$  ( $512/0.130 = 3938KB/s$ ). A separate set of experiments for dark fiber show that replay recovery lasts about 17% of the outage time in the transaction phase of the PostMark benchmark. I.e., a 60 second outage recovers in about 10 seconds.

The duration of full recovery is relative to the size of the logical volume. Table 8 shows that on a dark-fiber network, the full recovery transfer rate is 7.99–10.12MB/s. (Experiments show this rate to be independent of logical volume size.) If the SE recovers over a 51Mbps Internet link, the average full recovery rate is 2.9MB/s, which is similar to the replay recovery rate for the same configuration. When the bandwidth to the recovering SE increases, as seen before, the TCP window becomes the limiting factor.

Table 8 indicates that PostMark performance degrades more during full recovery than during replay recovery. The reason is that the data source for full recovery is an SE RAID that is also handling PostMark I/O, whereas replay recovery reads the contents of the replay log, which is stored on a separate disk.

## 7 Surprises and dead ends

Here are some of the noteworthy unexpected hurdles that we encountered during the development of StarFish:

- The required set of SCSI commands varies by device and operating system. For example, Windows NT requires the WRITE AND VERIFY command, whereas Solaris requires the START/STOP UNIT command, although both commands are optional.
- We spent considerable effort to reduce TCP/IP socket latency. We fixed problems ranging from intermittent 200ms delays caused by TCP's Nagle algorithm [22, section 19.4], to delays of several

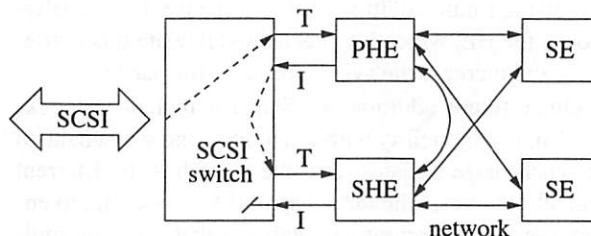


Figure 7: Redundant host element (RHE) architecture.

seconds caused by buffer overflows in our memory-starved Ethernet switch.

- Spurious hardware failures occurred. Most SCSI controllers would return error codes or have intermittent errors. However, some controllers indicated success yet failed to send data on the SCSI bus.
- For a long time the performance of StarFish with multiple SEs was degraded due to a shared lock mistakenly held in the HE. The shared lock caused the HE to write data on one socket at a time, instead of writing data on all sockets concurrently. This problem could have been solved earlier if we had a tool that could detect excessive lock contention.

There was a major portion of the project that we did not complete, and thus removed it from the released code. We have designed and implemented a redundant host element (RHE) configuration to eliminate the HE as a single point of failure, as depicted in Figure 7. It consists of a primary host element (PHE), which communicates with the host and the SEs, and a secondary host element (SHE), which is a hot standby. The SHE backs up the important state of the PHE, and takes over if the PHE should fail. We use a BlackBox SW487A SCSI switch to connect the PHE or SHE to the host in order to perform a transparent recovery of the host element without any host-visible error condition (except for a momentary delay). After spending several months debugging the code, we still encountered unexplained errors, probably due to the fact that the combination of the SCSI switch and the

target mode driver was never tested before. In retrospect, we should have implemented the redundant host element with a Fibre Channel connection instead of a SCSI connection, since Fibre Channel switches are more common and more robust than a one-of-a-kind SCSI switch.

## 8 Future work

Measurements show that the CPU in the HE is the performance bottleneck of our prototype. The CPU overhead of the TCP/IP stack and the Ethernet driver reaches 60% with 3 SEs. (StarFish does not use a multicast protocol, it sends updates to every SE separately.) A promising future addition could be using a TCP accelerator in the HE, which is expected to alleviate this bottleneck and increase the system peak performance.

Other future additions to StarFish include a block-level snapshot facility with branching, and a mechanism to synchronize updates from the same host to different logical volumes. The latter capability is essential to ensure correct operations of databases that write on multiple logical volumes (e.g. write transactions on a redo log and then modify the data).

The only hurdle we anticipate in porting StarFish to Linux is the target mode driver, which may take some effort.

## 9 Concluding remarks

It is known that the reliability and availability of local data storage can be improved by maintaining a remote replica. The StarFish system reveals significant benefits from a third copy of the data at an intermediate distance. The third copy improves safety by providing replication even when another copy crashes, and protects performance when the local copy is out of service.

A StarFish system with 3 replicas, a write quorum size of 2, and read-only consistency yields better than 99.9999% availability assuming individual Storage Element availability of 99%. The PostMark benchmark performance of this configuration is at least 85% of a comparable direct-attached RAID unit when all components of the system are in service, even if one of the replicas is connected by communication link with long delays and a limited bandwidth. During recovery from a replica failure, the PostMark performance is still 67–90% of a direct-attached RAID. For many applications, the improved data reliability and availability may justify the modest extra cost of the commodity storage and servers that run the StarFish software.

Although we report measurements of StarFish with a SCSI host interface, StarFish also works with a Qlogic ISP 1280 Fibre-Channel host interface.

The availability analysis in Section 4 assumes independent failures. However, a StarFish system may suffer

from *correlated* failures due to common OS or application bugs. One way to alleviate it is to deploy StarFish on a heterogeneous collection of servers from different vendors running different OSes (e.g. FreeBSD and Linux).

**Source code availability.** StarFish source is available from <http://www.bell-labs.com/topic/swdist/>.

**Acknowledgments.** We would like to thank our shepherd, Robert Watson, and the anonymous referees for their valuable comments. We would also like to thank Justin Gibbs for his help in the development of the Adaptec target-mode driver for FreeBSD and for his help in problem isolation, and Matthew Jacob for his help in enhancing the Qlogic target-mode driver for FreeBSD.

## References

- [1] AT&T. AT&T data & IP services: Backbone delay and loss, Mar. 2002. Available at [http://ipnetwork.bgtmo.ip.att.net/delay\\_and\\_loss.shtml](http://ipnetwork.bgtmo.ip.att.net/delay_and_loss.shtml).
- [2] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 79–86, Dec. 1985.
- [3] G. Bolch et al. *Queueing Networks and Markov Chains*. John Wiley & Sons, New York, 1998.
- [4] EMC Corporation. Symmetrix remote data facility product description guide, 2000. Available at [www.emc.com/products/networking/srdf.jsp](http://www.emc.com/products/networking/srdf.jsp).
- [5] E. Gabber et al. Starfish: highly-available block storage. Technical Report Internal Technical Document number ITD-02-42977P, Lucent Technologies, Bell Labs, April 2002.
- [6] G. Gibson and R. V. Meter. Network attached storage architecture. *Communications of the ACM*, 32(11):37–45, Nov. 2000.
- [7] IBM Corporation. IBM 99.9% availability guarantee program. Available at [www.pc.ibm.com/ww/eserver/xseries/999guarantee.html](http://www.pc.ibm.com/ww/eserver/xseries/999guarantee.html).
- [8] A. N. S. Institute. The SCSI-2 common access method transport and SCSI interface module, ANSI x3.232-1996 specification, 1996.
- [9] J. Katcher. PostMark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997. Available at [www.netapp.com/tech\\_library/3022.html](http://www.netapp.com/tech_library/3022.html).
- [10] N. Lawson. SCSI target mode driver. A part of FreeBSD 5.0-RELEASE, 2003. See `targ(4)` man page.

- [11] E. K. Lee and C. A. Thekkath. Petal: Distributed shared disks. In *Proceedings of the 7th Intl. Conference on Arch. Support for Prog. Lang. and Operating Systems*, pages 84–92, Oct. 1996.
- [12] E. Levy and A. Silberschatz. Distributed file systems: concepts and examples. *ACM Computing Surveys*, 22(4):321–374, Dec. 1990.
- [13] M. MacDougall. *Simulating Computer Systems*. The MIT Press, Cambridge, Massachusetts, 1987.
- [14] A. McEvoy. PC reliability & service: Things fall apart. *PC World*, July 2000. Available at [www.pcworld.com/resource/article.asp?aid=16808](http://www.pcworld.com/resource/article.asp?aid=16808).
- [15] W. T. Ng et al. Obtaining high performance for storage outsourcing. In *Proceedings of the USENIX Conference on File and Storage Systems*, pages 145–158, Jan. 2002.
- [16] B. Nicols, D. Buttlar, and J. P. Farrel. *Pthreads Programming*. O'Reilly & Associates, 1996.
- [17] H. Patterson et al. SnapMirror: File system based asynchronous mirroring for disaster recovery. In *Proceedings of the USENIX Conference on File and Storage Systems*, pages 117–129, Jan. 2002.
- [18] L. Rizzo. *dummynet*. Dipartimento di Ingegneria dell'Informazione – Univ. di Pisa. [www.iet.unipi.it/~luigi/ip\\_dummynet/](http://www.iet.unipi.it/~luigi/ip_dummynet/).
- [19] J. Satran et al. *iSCSI*. Internet Draft, Sept. 2002. Available at [www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-16.txt](http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-16.txt).
- [20] S. Savage and J. Wilkes. AFRAID – a frequently redundant array of independent disks. In *Proceedings of the Winter 1996 USENIX Conference*, pages 27–39, Jan. 1996.
- [21] D. P. Siewiorek. *Reliable Computer Systems: Design and Evaluation*. A K Peters, 1998.
- [22] W. R. Stevens. *TCP/IP Illustrated, Volume 1, The Protocols*. Addison-Wesley, 1994.
- [23] R. Stewart et al. *Stream Control Transmission Protocol*. The Internet Engineering Task Force (IETF), RFC2960, Oct. 2000. Available at [www.ietf.org/rfc/rfc2960.txt](http://www.ietf.org/rfc/rfc2960.txt).
- [24] L. Svobodova. File services for network-based distributed systems. *ACM Computing Surveys*, 16(4):353–368, Dec. 1984.
- [25] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, pages 305–318, Oct. 2000.





# Secure and Flexible Global File Sharing

Stefan Miltchev  
*miltchev@dsl.cis.upenn.edu*  
University of Pennsylvania

Vassilis Prevelakis  
*vp@drexel.edu*  
Drexel University

Sotiris Ioannidis  
*sotiris@dsl.cis.upenn.edu*  
University of Pennsylvania

John Ioannidis  
*ji@research.att.com*  
AT&T Labs – Research

Angelos D. Keromytis  
*angelos@cs.columbia.edu*  
Columbia University

Jonathan M. Smith  
*jms@dsl.cis.upenn.edu*  
University of Pennsylvania

## Abstract

Trust management *credentials* directly authorize actions, rather than divide the authorization task into authentication and access control. Unlike traditional credentials, which bind keys to principals, trust management credentials bind keys to the authorization to perform certain tasks.

The Distributed Credential FileSystem (DisCFS) uses trust management credentials to identify: (1) files being stored; (2) users; and (3) conditions under which their file access is allowed. Users share files by delegating access rights, issuing credentials in the style of traditional *capabilities*. Credentials permit, for example, access by remote users not known in advance to the file server, which simply enforces sharing policies rather than entangling itself in their management. Throughput and latency benchmarks of our prototype DisCFS implementation indicate performance roughly comparable to NFS version 2, while preserving the advantages of credentials for distributed control.

**Keywords:** Filesystems, access control, trust management, credentials.

## 1 Introduction

The Internet offers the possibility of global information sharing and collaboration. Existing file-sharing systems and their access control models have been challenged, however, by the scale and administrative complexity of the Internet. Such systems are either primarily distribution-oriented, such as the Web, or small in scale, such as NFS.

Both types of systems share the property that a (relatively) small set of users have read/write access to files, as current access control systems rely on authentication

requiring that the user is known to the system. This works in closed administrative domains (e.g., NIS domains or Kerberos realms [21]), where an *administrator* creates a user account and assigns access rights to it.

Consider a local user, Alice, who wishes to share files with Bob, who does not have an account on Alice's file server. The local system administrator must open an account for Bob, but this may create administrative and legal problems, and may conflict with local policies (e.g., only employees may have accounts).

We propose a mechanism that allows Alice, without the intervention of any centralized administrative authority, to authorize Bob to access her files. This is done by having Alice create a credential that contains Bob's key, the DisCFS file handle and the permissions. Alice signs the credential, and confers to Bob the authority to access the file. Alice may simply e-mail the authorization credential to Bob as cleartext because the credential itself does not contain any secrets (apart from the information that Alice wishes to share a file with Bob). By combining Alice's credential with one signed by himself, Bob may further delegate access to the file.

We show that this simple mechanism is secure and scalable. Further, by requiring the cooperation of only the users involved in the file exchange, this mechanism offers great flexibility and low administrative overheads. The system monitors all access to files, and can identify, using the offered public key, any entity issuing file requests. Mechanisms for restricting access or imposing access controls are also provided.

The access control mechanism that is presented in this paper is independent of the actual mechanism used for the exchange of data (e.g., ftp, NFS, http, and so on). We implement two prototypes, one for *ftp*-like access (not covered in this discussion) and another using the NFS protocol.

In the following sections we demonstrate how our mech-

anism may be deployed in practice using the NFS prototype as an example. We integrate our access mechanism with a user-level NFSv2 server using IPsec [16]; our intention is to offer this access mechanism eventually as part of the standard NFS authentication framework. The performance measurements collected by running common file-related benchmarks indicate performance roughly comparable to existing systems.

*Organization* The next section expands on the challenges addressed by DisCFS. Section 3 discusses prior and related work. Sections 4 and 5 describe our design and prototype implementation for OpenBSD 3.1, and Section 6 presents measurements of the prototype. Section 7 presents user and administrator experiences with our first prototype. Sections 8 and 9 discuss future work and conclude the paper.

## 2 Motivation

Let us consider two typical examples of information sharing. In the first case, Alice, a salesperson, would like her ten best clients to have access to advance information about a product. She does not want other clients or the general public to have access to this information, however.

The second example involves servers used to share information such as digital photographs (*e.g.*, `www.ofoto.com`). Alex is given the authority to store his personal photographs on a server. Apart from Alex, access to this information may be restricted to small groups of users. These groups may be different, depending on the material (*e.g.*, pictures of family events to relatives, pictures of social events only to those who participated, *etc.*).

In both cases, local users (Alice and Alex), known to their systems, wish to provide access to other external users. For this type of activity to be feasible, the following conditions must be met:

- The system should be able to cope with large numbers of files and an even larger number of users accessing these files.
- Administrator involvement to allow external users access to files should be eliminated. Local users should be able to authorize access to files by external users.
- The file access conditions must be flexible and expandable: there should be no constraints by the access mechanism as to what conditions may be imposed for access.

- Delegation is extremely important for the operation of the system. There is already an implicit delegation of access authority from the administrators to the local users and from the local users to external users, but we want to generalize the ability to delegate to arbitrarily long delegation chains.
- Most sites have access policies and these must be enforced regardless of the actions of individual users. The administrator should be able to specify default access policies for the entire site.
- Apart from the actual files, the system should maintain as little additional state as possible.
- The access mechanism should work for both centralized servers and in a distributed environment where the files are stored on multiple servers.

Existing systems have several major shortcomings when used for sharing information. First, traditional user authentication implies that a user is known to the system before file requests can be processed. Second, file and directory permissions are concepts inherited from multi-user operating systems. Sharing is achieved by either account sharing (which defeats accountability) or through the use of group access permissions on files and directories. Such permissions lack flexibility and fine granularity, and perhaps most importantly, extensibility: there is no way of adding new permission mechanisms if the existing ones prove inadequate.

In the salesperson example, because the information is not intended to be widely available, Alice must place the literature in a restricted part of the corporate Web site and make arrangements so that only the designated clients have access to the material. The traditional way of doing things implies that accounts and passwords should be created and given to the customers. A more sophisticated way of achieving the same goal is to use X.509 credentials for user authentication [7]. Although this approach addresses the well-known security problems of password authentication, it does not address the problem of access control, necessitating the maintenance of additional state (*e.g.*, access lists) on the server.

Faced with complex and inflexible mechanisms, some sites abandon access restrictions, and instead rely on obscurity (*e.g.*, non-obvious URLs, files in unreadable directories, *etc.*). Others use cookies, despite the fact that they are known to have numerous weaknesses [11].

Before we continue with the description of the Distributed Credential File System (DisCFS), which was designed and implemented to meet the listed requirements, we shall discuss previous work done in the area of wide-area file sharing.

### 3 Related Work

Network file sharing has attracted the attention due to the need for distributed information sharing has expanded with the reach and services of the Internet, and will continue to do so. There are some undesirable trade-offs in existing systems that inhibit large-scale network file sharing.

#### 3.1 File Systems

Network file systems such as NFS and AFS [27, 14] are the most popular and widespread mechanisms for sharing files in tightly administered domains. However, crossing administrative boundaries creates numerous administrative problems (*e.g.*, merging distinct Kerberos realms or NIS domains). The developers of the Athena, Hesiod and Bones systems recognize and address some of these problems [26, 8, 15].

Encrypting file systems such as CFS [4] place great emphasis on maintaining the privacy of the user information by encrypting the file names and their contents. The limitation of such systems is that sharing is particularly difficult to implement; the file owner must communicate the secret encryption key for the file to all the users who wish to access it. Even then, traditional access controls must still be used to enforce access restrictions (*e.g.*, read-only, append-only, immutable file, *etc.*). Furthermore, because CFS encrypts data stored in files, modifying files or altering their permissions is inefficient. Our system assumes that the server is trustworthy, so that the files can be stored in the clear. An administrator may still choose to deploy CFS-like encryption mechanisms on top of DisCFS.

WebFS is part of the larger WebOS [29] project at UC Berkeley. It implements a network file system on top of the HTTP protocol. WebFS relies on user-level HTTP servers, used to transfer data, along with a kernel module that implements the file system. The security architecture for WebOS, called CRISIS [3], uses X.509 certificates to identify users and to transfer privileges. Associated with each file are access control lists (ACLs) that enumerate which users have read, write, or execute permission on individual files. We have taken a more general and scalable approach in that there is no need for traditional ACLs because each credential is sufficient to identify both users and their corresponding privileges.

Bayou [28, 23] is a replicated, weakly consistent storage system, designed for the mobile computing environment. Like CRISIS, it uses certificates to identify users, and permits read or write access to data collections. It also supports delegation and revocation certificates. Dis-

CFS supports finer-grained access than Bayou; DisCFS credentials can contain much richer security policies.

The design rationale of the *capafs* system [24] is similar to that of DisCFS. Security for file access is done by encoding the access capabilities in the file name. This both results in incomprehensible file names (requiring symbolic links for user interactions) and knowledge of the file name providing access to the file (requiring protection of the file name and raising the question of how file names are communicated amongst remote users). In addition, limitations in the way directories are handled restrict file creation operations to local users.

The system that is most closely related to our work is the secure file system, or SFS [20]. SFS introduces the notion of *self-certifying pathnames* — file names that effectively contain the appropriate remote server's public key. In this way SFS needs no separate key management machinery to communicate securely with file servers. However, because SFS relies on a trusted third party to mutually authenticate server and client (or otherwise requires the use of a secure password protocol between them), collaboration is possible only if the client and the server have a common root for their Certification Authorities. DisCFS goes a step further. It uses credentials to identify both the files stored in the file system and the users that are permitted to access them, as well as the circumstances under which such access is allowed. Furthermore, users can delegate access rights simply by issuing new credentials, providing a natural and very scalable way of sharing information. This is not the case for SFS, where access control relies on user and group ID, which are translated from one machine to another. This forces users to have accounts on file servers to access protected files, and defeats the purpose of a truly distributed file system.

Putting DisCFS in the secure storage framework devised by Riedel, *et al.* [25], DisCFS has the following characteristics:

**Players.** As with iSCSI [10], there is no notion of individual users; readers, writers and owners are undifferentiated with respect to the credentials they possess.

Likewise, because access is not based on membership groups, there is no group server. There is no namespace server either, only a DisCFS storage server.

**Trust assumptions.** Data is not protected from the server, making it vulnerable to a collusive storage server, as with iSCSI.

**Security primitives.** Servers and hosts authenticate with mechanisms external to DisCFS. As with



iSCSI, any authentication keys are distributed by an external mechanism.

The server does differentiate between reads and writes. However, whether the server grants a read or write request depends solely on information contained in the credential presented by the requesting party.

Credentials provide appropriate access control and can be sent in the clear. For further security, data and commands can be encrypted while on the wire using IPsec.

Revocation is achieved by revoking credentials. Keromytis discusses a variety of mechanisms for credential revocation in trust management systems in [17].

**Granularity.** DisCFS is agnostic to the duration of the keys and credentials, but our expectation is that they are reasonably persistent.

**Convenience.** DisCFS is convenient to use. Once credentials for a file are available, delegation is easier than in identity-based systems, as we describe in some detail later in the paper.

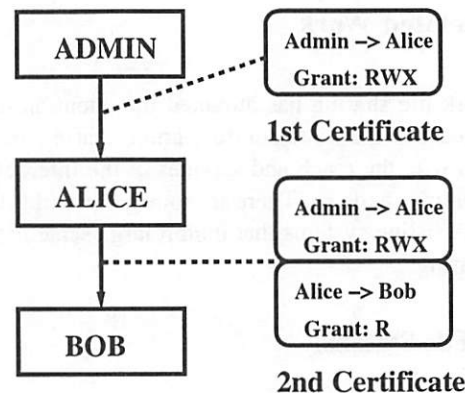
### 3.2 Operating Systems

The Taos operating system [30] uses a *narrow* API to control access to security-aware services, *e.g.*, the file system. It uses credentials for delegation of access rights between principals. DisCFS utilizes these well-known techniques to extend a real world protocol such as NFS. This makes our approach easily portable and more flexible than specialized operating system mechanisms.

The concept of credential-based access control also appears in the Exokernel [19]. In this system, users can create new capabilities at will, but the new capability must be dominated by an existing one. This is similar to our chains of certificates, but is limited by the fact that permissions are hardwired into the system, and the hierarchical capability tree may be only up to 8 levels deep. In our system, certificate chains can be of arbitrary length, and the access policy can consider factors such as time-of-day, so that, for example, leisure-related files may not be available during office hours.

### 3.3 Other Protocols

To share files across wide-area networks, a number of protocols have been deployed, the most commonly used ones being FTP and HTTP. Anonymous FTP, where there is no need for authentication, offers high flexibility



**Figure 1:** Delegation of privileges, from the administrator to Alice, and from Alice to Bob. The administrator grants Alice full access by issuing her the first certificate. Alice can then delegate read access to Bob by issuing him the second certificate. To be granted access Bob must present a certificate chain consisting of both certificates.

because any user can download or upload files to FTP servers. Similarly, in the Web architecture, access is either anonymous or subject to some sort of *ad hoc* authentication mechanism. This configuration is useful only in the case where file content is not sensitive. In the case where authentication is required, flexibility is greatly reduced. The only users allowed to access the server, in that case, are users who are already known to the system. As with existing network file systems, this restricts the possibility for users outside the same administrative domain to collaborate.

## 4 DisCFS Design

### 4.1 System Architecture

DisCFS dispenses with user names as an indirect means for performing access control; in other words, it does not require users to first authenticate to the system and then have their identities checked against access control lists in order to determine whether the server should honor a client's request. Instead, DisCFS uses a direct binding between a public key and a set of authorizations. This results in a decentralized authorization system that is flexible enough to cope with a large variety of authentication scenarios. Requests are signed with the requester's key and must be accompanied by other credentials that form a chain of trust linking the requester's key to a key that is trusted by the system. In our first example in Section 2, we looked at Alice's predicament in trying to allow her sales clients access to internal files. In the DisCFS sys-



tem, the server trusts only the administrator's key. An administrator signs Alice's credential, binding her key to the files in question. The credential allows Alice read, write and execute access to the files.

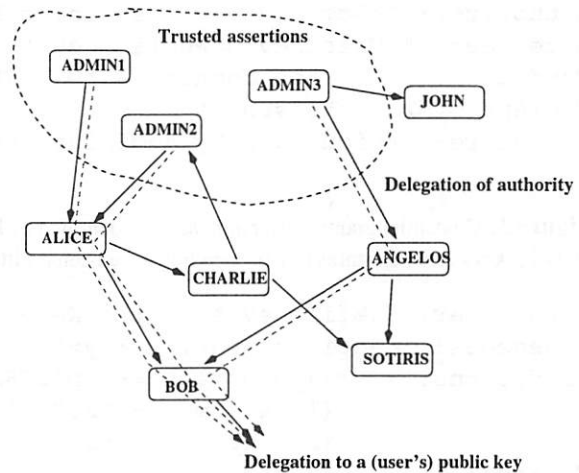
If Alice then wishes to allow Bob to read these files, she simply creates a new credential which grants Bob's key read access to the files. Bob issues a request signed with his key. If the system is to honor Bob's request, Alice's credential must accompany it. This credential forms a link between the external user (Bob) and the internal user (Alice). Alice's own credential (issued by the administrator) must also be available, to link the internal user to the administrator. Thus, Bob's request must be accompanied by both credentials in order to be granted (see Figure 1). Credential caching can reduce the number of credentials that have to be exchanged.

In DisCFS the traditional problem of credential (or certificate) revocation is fairly straightforward to address: because the DisCFS server controls access to files by examining a user's credentials, revocation is accomplished by notifying the server about bad keys or credentials. If the credentials are relatively short-lived, the server need only remember such information for a short period of time.

To express access rights and the diverse conditions under which these are granted, we need some form of policy definition language. There are a number of possible choices such as, PolicyMaker [6], KeyNote [5], QCM [12] and SPKI [9]. In our system we use the KeyNote trust management system for this purpose. Our choice was based on the fact that KeyNote is also integrated into IPsec which protects communication between the client and the file server. By using IPsec and Keynote, we can also use the file access credentials to establish the IPsec link (see Section 4.3).

## 4.2 Access Control in DisCFS

Generally speaking, access control systems check whether a proposed action conforms to policy. An administrator specifies actions as a set of name-value pairs, called an action attribute set. Policies written in the KeyNote assertion language either accept or reject action attribute sets that the system presents to the policy engine (non-binary results are also possible). An administrator can break up policies and distribute them as credentials, which are signed assertions that he can send over a network. A local policy can then refer to the credentials when making its decisions. The credential mechanism allows for arbitrarily complex graphs of trust, in which credentials signed by several entities are considered when authorizing actions, as seen in Figure 2. An administrator can handle group-based access



**Figure 2:** Delegation in KeyNote, starting from a set of trusted keys. The dotted lines indicate a delegation path from a trusted public key (the administrator's) to the user making a request. If all the credentials along that path authorize the request, it will be granted. Intermediate credentials can only refine the authority granted to them (*i.e., they can never allow a request that was denied by policy*).

simply by issuing the appropriate credentials to all the group members. Furthermore, an administrator can create sub-groups at any level in the hierarchy.

Figures 3 and 4 show two credentials that form part of a delegation chain in our system. The administrator issues a credential to user *miltchev* granting access to a particular directory. User *miltchev* then grants user *sotiris* access to the same directory during November 6, 2002. KeyNote allows this delegation because the authority granted to user *sotiris* is a subset of the authority of user *miltchev*. Notice that users are identified only by their public keys.

The advantage of using DisCFS is that an administrator no longer needs to have *a priori* knowledge of the user base. Thus, the system does not need to store information about every person or entity that may need to retrieve a file. DisCFS also provides its users with the ability to propagate access to the files by passing on (delegating) their rights to other users. In this way, users pass credentials rather than passwords, allowing the system to associate access requests with keys and to reconstruct the authorization path from the administrator to the user making the request (and thus grant access). The system may not know that Bob is trying to get at a file, but can log that key B (Bob's key) was used and that key A (Alice's key) authorized the operation. The administrator can then use this audit trail to validate that the access request follows the appropriate usage policy. Note

```

Authorizer: "<Administrator's Public Key>"
Licensees: "<Miltchev's Public Key>"
Conditions: (app_domain == "DisCFS") && (HANDLE == "666240") -> "RWX";
Comment: "testdir"
signature: "<Signature by Administrator>"

```

**Figure 3:** Credential granting user *miltchev* (as identified by his public key, in the *Licensees* field) access to directory *testdir*. The 1024-bit keys and signatures in hex encoding have been omitted in the interest of readability.

```

Authorizer: "<Miltchev's Public Key>"
Licensees: "<Sotiris' Public Key>"
Conditions: (app_domain == "DisCFS") && (HANDLE == "666240") &&
            (localtime >= "20021106000001") &&
            (localtime <= "20021106235959") -> "RWX";
Comment: "testdir"
signature: "<Signature by Miltchev>"

```

**Figure 4:** Credential by user *miltchev* granting (delegating) user *sotiris* access to directory *testdir* for one day. Again, the keys and signatures have been omitted in the interest of readability.

that a user can at most pass on the privileges she holds (there is no *rights amplification*); furthermore, the user can choose to delegate only a subset of her privileges, e.g., access only to a specific file in the user's directory.

### 4.3 DisCFS over NFS

We implemented DisCFS over NFS. This allows easy integration into existing systems without extensive modification. Moreover, the entire scheme works with both monolithic and distributed servers. Each DisCFS repository is responsible only for the part of the distributed filesystem that is stored locally, thus there is no need to distribute and synchronize authentication and access control databases (such as NIS).

The NFS protocol is particularly suitable for our needs for the following reasons:

- NFS is widely used and supported by numerous platforms.
- The NFS protocol is portable, stable and reliable.
- The NFS server is available as a user-level program, so development is possible without modifying the operating system. This is particularly useful because it is not always possible to have access to the operating system source code.

Like NFS, the DisCFS system consists of a client and a server. The client runs on the user workstation and establishes a connection to the DisCFS server. We use IPsec [16] to protect traffic between client and server.

The mutual authentication required for building an IPsec connection is based on the submitted file access credential (and additional delegation credentials). The client can authenticate the server, because the file access credential contains the server key, while the server only proceeds with the connection if the submitted credentials allow access to the requested file (thus establishing a chain of trust to the user's key).

When a file is stored in DisCFS, the server generates a credential containing information that allows the future retrieval of the file contents, as well as information about the file creator. Because DisCFS closely follows NFS semantics, it appears to the user as another mounted file system. Files for which credentials have been supplied appear under the mount point of the DisCFS file system. Without an appropriate credential, retrieval of a file is not possible.

Once a user submits the necessary file credentials, the file appears under the DisCFS mount point using the same name it had when its credential was created. The client may then use file I/O requests similar to NFS. The system also permits a user to override the default file name, allowing files to be placed in user-specified locations. This is because DisCFS access credentials allow direct access to files, making file naming optional. The name is stored as a comment in the credential and is used as the default file name. The operation of establishing a connection to the server is similar to the Unix *mount(8)* command whereby an entire filesystem is grafted to the file tree. See Section 5 for a detailed explanation of how users access files on a DisCFS server.

If additional files must be accessed from the same server, the existing IPsec connection is used. This optimization allows the cost of the IPsec connection establishment to be spread over requests for multiple files.

#### 4.4 Security Analysis

Our system allows users of a server to access their files securely over an insecure medium (Internet). Unlike ACL-based systems that authenticate users and then check their access rights, our system is concerned only with capabilities associated with a key. The system must, therefore, decide whether a request signed with a given key should be granted or denied. The decision depends on whether the system can form a chain of trust between the issuing key and another key that the server can trust (*e.g.*, the key of a system administrator).

The chain of trust is formed by merging information that is pre-stored in the server (default policy) and policy statements that the user supplies. These policy statements are encapsulated in credentials. Each credential makes some assertions about one key and are signed with another key. All these assertions together with the authorizing keys are fed to the policy engine in the access control system.

Credentials are signed but not encrypted. They contain policy assertions and the public component of the key that they apply to. Both key and policy are signed with the authorizing key. The implication of this feature is that credentials may be retrieved on demand. For example, if Alice wants to send Bob a pointer to some information, instead of sending a URL to the relevant page, she may send Bob a URL pointing to the credentials needed to access the information. Because these credentials contain a reference to the file, Bob needs no further information to access the file. Obviously, using the http protocol to download credentials is one of many ways of acquiring them.

If such a credential is intercepted, the only information that may be obtained is that key A makes some assertions regarding key B. Credentials are signed and therefore cannot be forged. Because they relate to specific files (defined by the assertions that they contain), they cannot be used to obtain access to other information.

The default policies may define the rules that apply to a particular server. For example, they may allow access only between certain hours of the day, or they may exclude or limit access granted to a particular user.

Data is encrypted only while in transit. It is stored as cleartext on the server, which implies that users trust the server and its administrators. Existing data-protection schemes (such as a CFS-like filesystem) can be used on

top of DisCFS to protect the secrecy of the user's data from the server.

Credential-based access control systems usually have problems handling revocation, as it is difficult, if not impossible, to know who may have access to a file. However, by controlling the file server, administrators have a number of ways for disallowing access to files:

- Because the initial access credential uses the DisCFS file handle to address the corresponding file, an administrator can invalidate the access credential by changing the handle (see Figure 3).
- Administrators can disallow particular keys from being used (thus revoking the keys, rather than the credentials that grant them privileges) by modifying the site's security policy. This approach, however, is not scalable and should be used in conjunction with relatively short lived credentials (*i.e.*, with expiration periods measured in weeks, rather than years).
- Administrators can tie the access credentials to short-lived "refresher" credentials. Thus the access credential is not useful without a valid (non-expired) refresher.

Note that delegation, although extremely useful, can be turned off. Administrators can also limit the maximum length of the delegation chain (by inserting policy code in the access credentials), thus restricting the spread of delegated credentials.

The main advantage of having policy embedded within the credentials is that administrators can have multiple schemes operating at the same time. Different schemes may be used on different categories of files depending on, *e.g.*, the security classification of each category. Thus, for restricted files (lowest classification) administrators may rely on the expiration of the credentials; on more sensitive files they can use the default site policy, and if a file has to be unconditionally removed, administrators can change its handle and issue new credentials to the users that should access it.

#### 4.5 Threat Analysis

At the object level, the threat model of DisCFS does not seem any different from any simple file access control protocol. DisCFS does not encrypt files on disk, thus users have the option of trusting the system administrator or using encryption mechanisms on top of DisCFS.

At first glance the threat model of DisCFS at the network level does not differ from that of NFS used over a



secure channel, *e.g.* a trusted LAN or VPN. However, what sets DisCFS apart is its ability to address the threat of implicit rights amplification inherent in identity-based access control. All co-authors writing an arbitrary paper using CVS need to have login access to the serving machine. In contrast, the credential-based access control of DisCFS allows us to trust the co-authors no more than we have to - for authorship and nothing else.

## 4.6 Scalability

An important advantage of the access mechanism we present is scalability. This is due to the fact that we do not maintain user access rights on the DisCFS server. In fact, the server does not even have the concept of a “user” in the traditional operating system sense; it merely processes requests from keys that can supply a valid trust chain to one of the keys contained in the default policy of the server.

This design decision has two implications: (a) users must supply (sometimes long) chains of credentials, and (b) the server’s trust state remains constant irrespective of the number of potential users. Caching alleviates some of the processing overhead associated with the long trust chains.

Keeping servers uncontaminated by user information allows data set partitioning and replication across systems even in separate administrative domains. The disk capacity of the server must be proportional to the amount of information it contains, and its connection to the Internet should be appropriate for the actual traffic it experiences, not the user set or the number of policies that must be enforced.

Although the number of *potential* users is irrelevant, the number of *actual* users that connect to the server to access information is important because the access control operations they initiate load the server. It is clear that a KeyNote-based access control mechanism places greater load on the processing resources of the server, however this can be addressed in two ways: (a) by using hardware acceleration for the cryptographic operations, and (b) by replicating or partitioning datasets across servers, thus spreading the load over many access points.

## 5 Implementation Details

The DisCFS prototype was implemented on OpenBSD 3.1 by creating a user-level NFS server with the access control mechanism described in Section 4. The implementation is portable and does not depend on features unique to OpenBSD.

Administrators must set up initial NFS mountpoints on each client for each server that offers DisCFS services. To construct a secure path to the DisCFS server, the client constructs an IPsec tunnel between the client system and the DisCFS server. We extended the NFS protocol with an RPC procedure that enables the client to attach the remote directory to the mountpoint over the IPsec connection. This allows the DisCFS server to retrieve the public key used for authentication in the IKE [13] protocol (as part of the IPsec key establishment phase) and associate it with a Unix-style *userid*. IPsec protects future NFS requests, allowing the DisCFS server to associate them with the user’s public key. Each user on a multi-user client has his own IPsec connection to the server.

After a user executes our RPC attach procedure, the desired directory appears under the client’s default DisCFS mount point (*e.g.*, */discfs*). However, because the user has not yet provided any credentials, the file permissions of the attached directory are set to deny all access. File/directory ownership is set to the *userid* provided by the attach procedure. The *userid* is irrelevant to the DisCFS server, and thus no prior arrangement with the system administrator is needed. Similarly, no file ownership conflicts are possible; the *userid* is only manipulated in this way to make possible the use of unmodified NFS clients.

To get access to the attached directory or any other files/directories in it, the user must have a credential like the one shown in Figure 3. This credential is issued by the administrator (as identified by the public key appearing in the Authorizer field) to a specific user (as identified by the public key appearing in the Licensees field), and contains enough information for the DisCFS server to determine what permissions should be granted to the client system.

A file/directory is identified by a *handle*, which in our prototype implementation is simply the i-node number of the file/directory on the server. A DisCFS server uses this handle to locate the physical file in its local file storage. The handle specifics should change in the future because inode numbers are not suitable as globally unique identifiers across a network. A possible solution would be to build a handle from the inode number and a *generation number*, similar to the 4.4 BSD NFS implementation. In the following discussion we refer to file handles used by DisCFS as *DisCFS file handles* to distinguish them from NFS file handles.

The “Conditions” field can contain additional restrictions *e.g.*, allowing access during working hours, specifying that multiple authorizers are required, *etc.* By combining credentials with different policy assertions we construct a rich access framework that allows fine



grained access control.

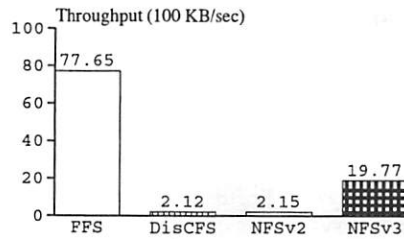
The credential assertions in our implementation grant standard Unix permissions. The return values for the assertions form a partial order of 8 combinations (“false”, “X”, “W”, “WX”, “R”, “RX”, “RW” and “RWX”) and translate directly into the standard octal representation. Thus, in the credential of Figure 3 the user is granted read, write, and execute access on the *testdir* directory. We wrote a utility which allows a user to submit credential assertions to the DisCFS daemon over RPC. The DisCFS server adds credential assertions to a persistent session. Following this operation, the permissions of the attached directory change accordingly. To improve performance, we use a simple round-robin cache of requested operations and policy results.

The semantics of some of the procedures defined by the NFS protocol change in our implementation. For example, because access control is managed through credential assertions, it makes no sense to use the *setattr* procedure for setting mode bits on a file. There is also a problem with the *create* and *mkdir* procedures. A user can create a file in the attached directory because he has read, write, and execute access. However, he cannot access the newly created file because he does not have a credential assertion for it. Thus, we added our own RPC procedures to the NFSv2 protocol that, upon successful creation of a file/directory, return a credential with full access to the creator of the file. Furthermore, this credential is also added to the user’s active session, so he can immediately use the newly-created filesystem object. The owner can then issue other credentials further delegating access to this file/directory to other users.

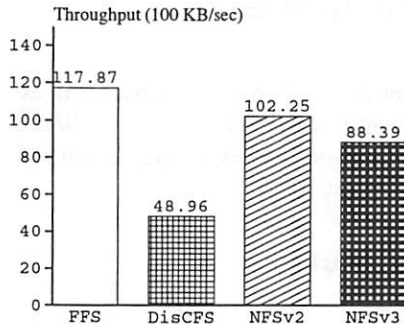
## 6 Experimental Evaluation

The architectural discussion is largely qualitative, and consequences for system performance are useful to understand. The best evaluation would be application-oriented benchmarks for applications in distributed environments, however this is always difficult for a new prototype system. Instead, we use micro-benchmarks and macro-benchmarks to obtain first-order quantification of performance, as well as identification of overhead introduced by the access control mechanism.

Our test hosts are 1 GHz Intel PIII machines with 256 MB of memory and 10 GB Western Digital Protege IDE hard drives. In the two-host, client-server tests that explore the network performance of our system, we connect our machines with 100 Mbps Ethernet. We did not use IPsec for the measurements, because results would vary considerably depending on the specific IPsec con-



**Figure 5:** Bonnie Sequential Output (Block). DisCFS throughput is comparable to NFSv2 throughput.



**Figure 6:** Bonnie Sequential Input (Block). DisCFS throughput is less than 50% of NFSv2 throughput, mostly due to its lack of prefetching.

figuration parameters. For an evaluation of the performance impact of using IPsec in various configurations we refer the reader to [22]. In the following tables, *FFS* means measurements taken on the local file system. *NFSv2* and *NFSv3* denote measurements over NFS protocol versions 2 and 3 respectively using the UDP protocol. We did not measure our performance when running over TCP.

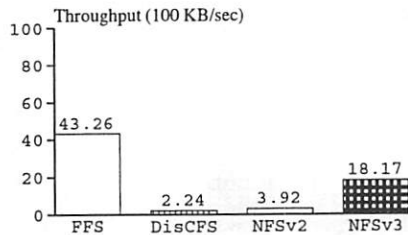
### 6.1 Micro-benchmarks

We use the *Bonnie* benchmark [1] to evaluate performance when writing and reading a large file. To eliminate caching effects we use a 512MB file, twice the size of main memory. In our results we also include the performance of *FFS* and *NFSv3* for completeness.

Figure 5 presents results for block writes. The performance of *DisCFS* is equivalent to that of *NFSv2* because the credential related overheads of *DisCFS* are amortized over the time of the experiment. Remember that this experiment is on a single file.

Figure 6 presents results for block reads. We observe that for read operations *DisCFS* trails *NFSv2* by more than 100%. The reason for this behavior is that *NFSv2* prefetches blocks, an optimization which we have yet to incorporate in the current implementation of *DisCFS*.

Finally, in Figure 7 we experiment with the cost of rewriting blocks. More specifically, in this test Bonnie



**Figure 7:** Bonnie Sequential Output (Rewrite). DisCFS throughput is closer to NFSv2 throughput than for reads, but still suffers from its lack of prefetching.

reads a block, dirties it, and then writes it back. As expected, the performance of *DisCFS* is closer to *NFSv2* than in the read benchmarks, however *DisCFS* still suffers from its lack of prefetching.

## 6.2 Macro-benchmarks

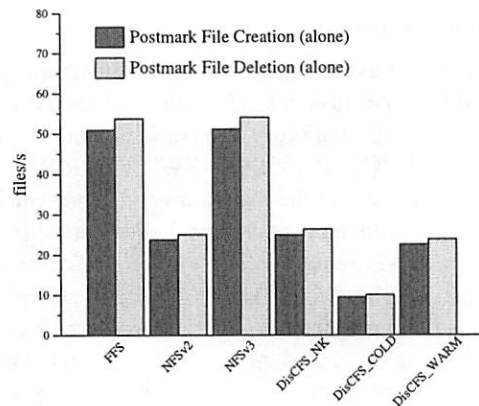
The Bonnie micro-benchmark gives us an idea about the raw performance of DisCFS. However, it does not reflect the access pattern typical of most modern applications. Internet software like electronic mail, netnews and web-based commerce depends on a large number of relatively short-lived files.

To simulate heavy small-file system loads we use the *PostMark* benchmark [2]. *PostMark* was designed to create a large pool of continually changing files and to measure the transaction rates for a workload approximating a large Internet e-mail server.

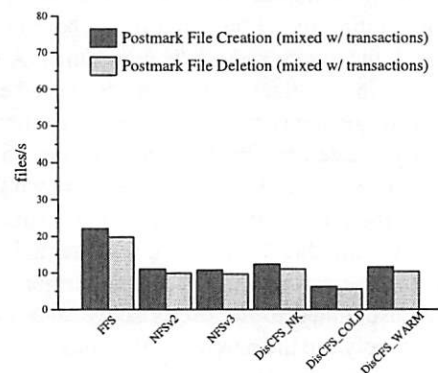
We use the default *PostMark* configuration parameters. The initial number of files created is 500. Files range between 500 bytes and 9.77 kilobytes in size. *PostMark* then performs 500 transactions on each file. Block sizes for reads and writes are 512 bytes and UNIX buffered I/O is used. We run each *PostMark* test 10 times and take the average.

We compare three versions of DisCFS to *FFS*, *NFSv2* and *NFSv3*. *DisCFS\_NK* is a crippled version of our system offering no security; no KeyNote queries are made. Instead, full access is returned for every file. *DisCFS\_COLD* is a fully functional system, but the server was restarted between each successive run of the benchmark. Results for *DisCFS\_WARM* reflect the effects of using a cache of 1024 policy results and not restarting the server between successive runs of the benchmark.

Figure 8 shows results for the average creation rate (files/second) for files created before other transactions were performed and the average deletion rate (files/second) for files deleted after all other transactions were performed. When *PostMark* creates a file, it selects a random initial size, and writes text from a random pool up to the chosen length. File deletion selects a random



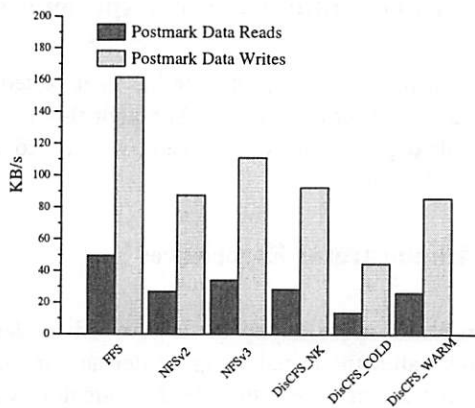
**Figure 8:** PostMark Average File Creation/Deletion Rate. Without the credential processing overhead, DisCFS performance is comparable to NFSv2. Full KeyNote functionality reduces performance by more than half. If requests are served from a warm cache, the performance is close to NFSv2 again.



**Figure 9:** PostMark File Creation/Deletion Mixed w/ Transactions. Performance drops when file create/delete operations are mixed with other transactions. However, DisCFS performance remains the same in relation to NFSv2.

file from the list of active files and deletes it. The performance of *DisCFS\_NK* is approximately equivalent to *NFSv2* when the credential processing overhead is eliminated. *DisCFS\_COLD* results show that the credential processing overhead is significant. Performance drops by more than 50%. This is not surprising because upon each file creation the DisCFS server must create a new credential, sign it and evaluate it in the KeyNote session. Upon each deletion a query must determine whether the operation should be permitted. As the DisCFS server begins to service most requests from the cache after the first run of *PostMark*, numbers for *DisCFS\_WARM* return to just below the performance of *DisCFS\_NK*.

Figure 9 also presents average file creation and deletion rates, but in this case the files are created and deleted during a sequence of other transactions. As expected,



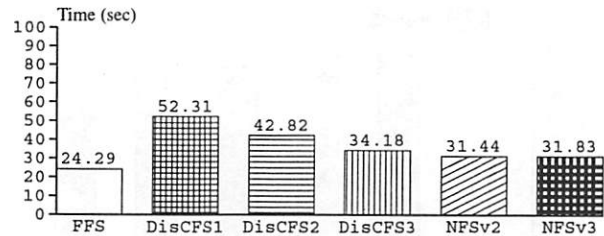
**Figure 10:** PostMark Data Read/Write Throughput. Throughput for writes is better than for reads due to the buffer cache. DisCFS performance in relation to NFSv2 performance remains the same as in Figures 8 and 9.

overall performance drops compared to the previous isolated case, but *DisCFS* performance remains the same in relation to *NFSv2* performance.

Finally, results presented in Figure 10 reflect system data throughput. For the read test, PostMark opens a randomly selected file and reads the entire file into memory using the configured block size (512 bytes). For the append test, PostMark opens a random file, seeks to the end of it, and writes a random amount of data. As expected, the throughput for writes is better than for reads because writes go through the buffer cache. Performance of *DisCFS* is again comparable to *NFSv2* when the credential overhead is eliminated artificially (*DisCFS\_NK*), or by caching (*DisCFS\_WARM*). With a cold cache (*DisCFS\_COLD*) the performance drops by more than half due to the frequent KeyNote queries.

We use the `top` utility to monitor CPU utilization during the PostMark benchmarks. The *NFSv2*, *NFSv3* and *DisCFS\_NK* servers utilize less than 1% of the CPU. Utilization for *DisCFS\_COLD* reaches up to 60% during the file creation test due to the number of cryptographic operations the server must perform. Caching brings the number down to 4% for *DisCFS\_WARM*.

To explore the overhead of credential handling imposed by real world applications, we time a recursive `grep` for `ifdefs` in every file of the OpenBSD kernel source tree. We conduct the test with a cache size of 128 policy results and a pool of 5000 sessions. The cache contains the permission bits returned by previous policy lookups. The *DisCFS* server adds credential assertions to a session. Having more sessions helps to distribute them and speeds up the query evaluation. We use three versions of our system: *DisCFS* with full credential functionality (*DisCFS1*), *DisCFS* with no signature verification of the

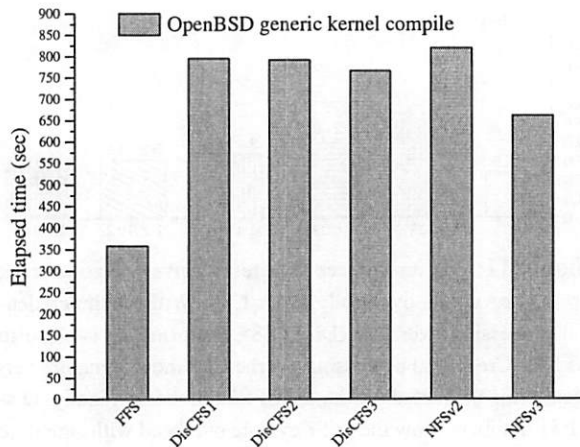


**Figure 11:** Recursive Grep. The results give us a good break up of the various overheads of *DisCFS*. Without the credential processing overhead (*DisCFS3*), performance is close to *NFSv2*. Credential processing overhead without signature verification is presented by the *DisCFS2* numbers. Finally, *DisCFS1* numbers show the full KeyNote overhead with signature verification.

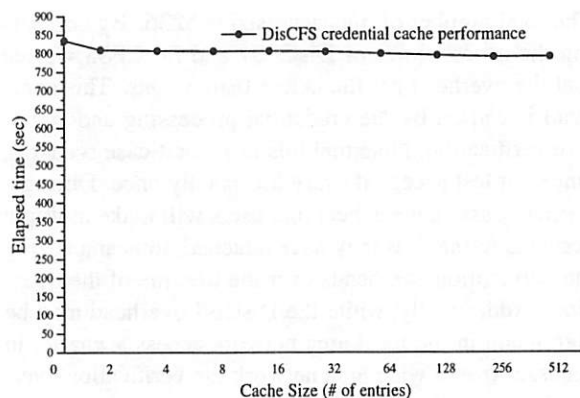
credentials (*i.e.*, all credentials are trusted) (*DisCFS2*), and finally a version of *DisCFS* that does not use credentials at all and just returns full access for every file (*DisCFS3*). We summarize our results in Figure 11.

The total number of files accessed is 5236. By comparing the access times of *DisCFS1* and *DisCFS3* we see that the overhead per file is less than 3.5 *ms*. This overhead is caused by the credential processing and signature verification. Note that this is a worst-case scenario, since our test accessed every file exactly once. During a normal session, we expect that users will make multiple accesses to the files they have attached, thus amortizing the verification overheads over the lifetime of their session. Additionally, while the *DisCFS* overhead may be significant in the local area network access scenario, in accesses over a wide area network the verification overheads become less significant.

As a more representative test, we compare the time to compile the OpenBSD kernel over the local filesystem (*FFS*), *NFSv2*, *NFSv3*, and the three versions of *DisCFS*. This experiment involves access control decisions for approximately 4500 source and 2600 generated files (object and header files, as well as the produced kernel image). We show our results in Figure 12. As expected, the local filesystem is the fastest; however, the cost of the full *DisCFS* implementation (including signature verification and complete policy evaluation per file access) is negligible, compared to the plain *NFSv2* case. (Notice that *DisCFS* seems to be slightly faster than *NFSv2*. The primary reason for this is the greatly simplified code running on the *DisCFS* server, which is effectively a very minimal *NFS* server.) The overhead due to credential signature verification, *i.e.*, the difference between *DisCFS1* and *DisCFS2*, is just 4 seconds. The overhead of using credentials, *i.e.*, the difference between *DisCFS1* and *DisCFS3* is about 30 seconds. The access control costs that are more evident in Figure 11 are amortized over the actual operation of the system. This conclu-



**Figure 12:** Compilation of the OpenBSD kernel. In this experiment the overheads due to credential evaluation and signature verification have been amortized over the actual operation of the system. Thus, the difference between DisCFS1 and DisCFS3 is only about 30 seconds.



**Figure 13:** DisCFS credential cache performance. Increasing the cache size results in gradual improvement.

sion matches our previous experience in evaluating the relative costs of security mechanisms: although cryptographic (or other) costs may seem high when viewed in the context of a micro-benchmark, their impact during regular user activities is minimal [22].

In our last experiment, we evaluate how different cache sizes affect DisCFS performance in our compilation experiment. In Figure 13, we see that increasing the cache size results in a gradual improvement, leveling off at about 256 entries. Even limited caching of policy decisions improves performance by more than 5%. Inevitably this will vary depending on the file access patterns and an extensive evaluation of optimum cache size is beyond the scope of this paper. However, we believe that it is beneficial to include even a small cache for keeping recent policy results.

## 7 User and Administrator Experiences

So far, our first DisCFS prototype has been tested in a small laboratory environment. Although this is not a wide-scale deployment by any means, we learned some important lessons.

### 7.1 Administrator Experiences

Administrators were happy to be relieved of dealing with users after the initial setup. After an administrator granted a user access to a desired directory with a signed credential, there was little reason for any user to contact an administrator. Further files could be created, and access could be delegated to other users without external administrative intervention. However, initial setup of the system was at times cumbersome. As our first DisCFS prototype was based on NFS, administrators had to deal with setting up initial NFS mountpoints on each client for each server that offered DisCFS services. In hindsight, it would have been better to circumvent the *mountd* protocol. Setting up DisCFS initially also required a good understanding of IPsec configuration.

Administrators also pointed out some security concerns they had with the first prototype. Currently, a KeyNote query is not performed for every nfs call, only on *getattr* calls. While this improves performance, it also means that we trust the client software to enforce the returned UNIX permission bits, something that will change in future releases of DisCFS.

### 7.2 User Experiences

Users initially thought that creating and signing credentials using a text editor and the KeyNote command-line utilities was somewhat cumbersome (*e.g.*, great care had to be taken with whitespace, so that signatures would be correct). This was quickly addressed with a Perl script that streamlined the process.

Sending credentials for each file over to the server proved to be inconvenient when large numbers of files were involved. A solution offering more end-user transparency is desirable.

## 8 Future Work

User and administrator experiences with the first prototype of DisCFS suggest that improvements can be made both in ease of use and security. We have started work on a second prototype which should address most issues.



We will eliminate the requirement to administer NFS mount points by including the name of the server hosting a file in the credential. It will then not be necessary to use the *mountd* protocol in order to determine a file's location. KeyNote queries will be made on all NFS calls, so that we do not have to trust the client software.

As mentioned previously, the choice of inode numbers as file handles was not optimal. Consider the case where Bob creates and shares file *song.mp3* with Alice. Bob soon gets tired of the song, deletes it, and proceeds to create another file, *tax\_return.dat*. If the inode of *song.mp3* happened to be used for *tax\_return.dat*, Alice might be granted access beyond the initial intent. To improve security, the new file handles will not use inodes.

To simplify semantics, our second prototype will not be used for sharing directories. The next version of DisCFS will support sharing of files only. Users will be able to choose what directory structure they want the files to appear in. Manual sending of credentials will be eliminated by using a loopback NFS server or a stackable filesystem layer on the client. When a file containing a credential is encountered (e.g., *Makefile.cred*), the loopback server will translate the name (*Makefile*), read the credential, connect to the remote server specified in the credential, and finally serve the file if the remote server grants access. While this additional layer will hurt performance somewhat, it will greatly enhance the transparency of DisCFS to the end user.

For our first prototype we chose a user-level NFS server because it made development a lot faster. Our benchmarks suggest that the additional context switching overhead compared to a kernel-level server is not substantial. In the future we might try to move the DisCFS server into the kernel which would likely bring about a small improvement in performance at the expense of portability. However, we suspect that larger benefits can be obtained by incorporating some of the NFS optimizations not incorporated in DisCFS (e.g., block prefetching).

DisCFS is based on NFSv2 because we had convenient examples of userland servers implementing it. It would be worthwhile to integrate our authentication and authorization model with NFSv4, e.g. as a plugin into the GSS-API [18].

We used a simple round-robin cache of policy results in our first prototype. We would like to implement a practical cache replacement algorithm and experiment more to find an optimal cache size.

Unix permission bits can be inflexible and limiting. Future versions of DisCFS could take advantage of the flexibility of credentials to provide more fine-grained access control, e.g., allow appending to a file but no truncation.

Looking further forward, new file-sharing policies are

achievable with DisCFS beyond the Unix NFS support demonstrated. An example of such use would be enabling controlled access to file storage for the untrusted users that are characteristic of the Web. DisCFS should also offer advantages for emerging P2P systems, as its avoidance of centralized control is well-suited to cooperative resource-sharing at a large scale and with disparate administrative models. Finally, while DisCFS's decentralized control should result in "scalability", this assertion requires robust quantitative modeling and supporting measurement.

## 9 Conclusions

This paper introduced a completely credential-based mechanism for authentication and access control of files. This is a new approach to distributed file systems, as it separates the *policy* for controlling the file from the access control *mechanism* used by the underlying file storage. This gives DisCFS advantages in flexibility and scalability over traditional file systems, and even over some recent secure file system efforts.

The DisCFS prototype implementation combines a credential-based access control system with common Unix file operations. It is straightforward to implement and deploy DisCFS because it uses components that exist in common operating systems, such as NFS and IPsec, and supports the traditional Unix filesystem semantics.

The system's performance was evaluated with both micro- and macro-benchmarks. The performance of individual DisCFS operations is bounded by that of the same primitives, such as remote RPC times, which limit the performance of other distributed systems. Used in larger contexts such as software builds or file-tree searches (where many files are "touched" sequentially, a worst case for DisCFS) the performance impact of DisCFS's enhancements is relatively low. In normal usage, the DisCFS-imposed overhead is negligible.

DisCFS source code is available for download at <http://www.seas.upenn.edu/~miltchev/discfs/>.

This work was supported by DARPA and NSF under Contracts F39502-99-1-0512-MOD P0001, CCR-TC-0208972, and CISE-EIA-02-02063.

## References

- [1] Bonnie Filesystem Performance Benchmark. <http://www.textuality.com/bonnie/>.
- [2] PostMark Filesystem Performance Benchmark. [http://www.netapp.com/tech\\_library/3022.html](http://www.netapp.com/tech_library/3022.html).
- [3] Eshwar Belani, Amin Vahdat, Thomas Anderson, and Michael Dahlin. The CRISIS Wide Area Security Architecture. In *Proceedings of the USENIX Security Symposium*, pages 15–30, August 1998.
- [4] M. Blaze. A Cryptographic File System for Unix. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, November 1993.
- [5] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust Management System Version 2. RFC 2704, September 1999.
- [6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the 17th Symposium on Security and Privacy*, pages 164–173, 1996.
- [7] CCITT. X.509: The Directory Authentication Framework. International Telecommunications Union, 1989.
- [8] S. P. Dyer. The Hesiod Name Server. In *Proceedings of the USENIX Winter Technical Conference*, pages 183–190, 1988.
- [9] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. RFC (Proposed Standard) 2693, Internet Engineering Task Force, September 1999.
- [10] J. Satran et al. IPS Internet Draft - iSCSI. Internet Draft, Internet Engineering Task Force, September 2001.
- [11] K. Fu, E. Sit, and N. Faemster. Dos and Don'ts of Client Authentication on the Web. In *Proceedings of the USENIX Security Symposium*, pages 251–269, August 2001.
- [12] C.A. Gunter and T. Jim. Policy-directed certificate retrieval. *SP&E*, 30(15):1609–1640, 2000.
- [13] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC (Proposed Standard) 2409, Internet Engineering Task Force, November 1998.
- [14] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [15] J. Schönwälder and H. Langendörfer. Administration of large distributed UNIX LANs with BONES. In *Proceedings of the World Conference On Tools and Techniques for System Administration, Networking, and Security*, April 1993.
- [16] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC (Proposed Standard) 2401, Internet Engineering Task Force, November 1998.
- [17] A. D. Keromytis. *STRONGMAN: A Scalable Solution to Trust Management in Networks*. PhD thesis, University of Pennsylvania, December 2001.
- [18] J. Linn. Generic Security Service Application Program Interface, Version 2. RFC (Proposed Standard) 2078, Internet Engineering Task Force, January 1997.
- [19] D. Mazieres and M.F. Kaashoek. Secure Applications Need Flexible Operating Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, May 1997.
- [20] D. Mazieres, M. Kaminsky, M. Frans Kaashoek, and E. Witchel. Separating key management from file system security. In *Symposium on Operating Systems Principles (SOSP)*, pages 124–139, December 1999.
- [21] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos Authentication and Authorization System. Technical report, MIT, December 1987.
- [22] S. Miltchev, S. Ioannidis, and A.D. Keromytis. A Study of the Relative Costs of Network Security Protocols. In *Proceedings of the Annual USENIX Technical Conference, Freenix Track*, pages 41–48, June 2001.
- [23] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: Replicated Database Services for World-Wide Applications. In *Proceedings of the 7th ACM SIGOPS European Workshop*, 1996.
- [24] J. Regan and C. Jensen. Capability File Names: Separating Authorization from User Management in an Internet File System. In *Proceedings of the USENIX Security Symposium*, pages 211–233, August 2001.
- [25] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST)*, January 2002.
- [26] M. A. Rosenstein, D. E. Geer Jr., and P. J. Levine. The Athena Service Management System. In *Proceedings of the Winter USENIX Conference*, pages 203–212, 1988.
- [27] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer USENIX Conference*, June 1985.
- [28] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, December 1995.
- [29] A. Vahdat. *Operating System Services for Wide-Area Applications*. PhD thesis, UC Berkeley, December 1998.
- [30] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos Operating System. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

# The CryptoGraphic Disk Driver

Roland C. Dowdeswell  
*elric@imrryr.org*  
The NetBSD Project

John Ioannidis  
*ji@research.att.com*  
AT&T Labs – Research

## Abstract

We present the design and implementation of CGD, the CryptoGraphic Disk driver. CGD is a pseudo-device driver that sits below the buffer cache, and provides an encrypted view of an underlying raw partition. It was designed with high performance and ease of use in mind. CGD is aimed at laptops, other single-user computers or removable storage, where protection from other concurrent users is not essential, but protection against loss or theft is important.

## 1 Introduction

The number of laptop users is increasing continuously, and with it the amount of sensitive data that resides outside traditional data protection mechanisms—physical (security guards) or electronic (firewalls). There have been several well-publicized cases of top executives having their laptops lost or stolen, with highly sensitive corporate data in them [3]. In fact, any potentially removable storage suffers from such threats, as the case of the missing disk drives from a government research lab around 2000 [1] demonstrates. Some of the cases involved the violation of doctor–patient privilege [10] which may expose doctors or hospitals to liability for disclosing confidential patient data. When computers are retired, the hard drives may contain sensitive data, complicating their disposal. Cryptography provides the ultimate protection in such cases, but while there exist many techniques and systems that employ cryptography to protect files, many of them are cumbersome to use, require expertise beyond that of the common user, or (unfortunately, in the case of many commercial systems) offer less security than advertised. We review some of these approaches in Section 4.

There are many existing solutions to this problem, such as CFS[4], TCFS[5], OpenBSD's vnd(4) encryption, FreeBSD's GEOM encryption, Linux's loopback encryption, Cryptfs[17] and NCryptfs[16]. CFS and TCFS operate over the file system layer and for that reason are well suited to encrypting files stored over a distributed file system such as NFS. CFS is implemented as a userland NFS server and gains much in the way of portability, but unfortunately gains all of the drawbacks of NFS in the process. The semantics of NFS even on the local machine make file locking less reliable, make

caching of data blocks less effective and have a serious overall impact on performance. CFS also does not protect the filesystem metadata as it encrypts each file separately and does not encrypt file system meta-data. It may be possible in many scenarios to get much of the information that is desired simply by examining the directory layout. TCFS also uses NFS as its transport, but operates as an NFS client rather than a server. TCFS has many more features than CFS, but also shares the drawbacks of using NFS.

To fulfill the needs for secure file storage in a laptop environment, we built the CryptoGraphic Disk (CGD). CGD is a device driver that looks just like an ordinary disk drive, and encrypts an entire real disk partition. It is thus suited for single-user environments (mostly laptops) where protection from other users on the same machine is not essential, but protection against physical loss is paramount, as is high performance. CGD was written for and is available in NetBSD, but can be readily ported to any other BSD-derivative with very few changes.

This is a similar approach to that taken by OpenBSD's vnd(4) driver, FreeBSD's GEOM driver and Linux's loopback mounts. Each of these has a few drawbacks. OpenBSD's vnd(4) driver and FreeBSD's GEOM driver do not present a modular framework for defining encryption methods which we discuss in Section 2. We discuss some of the drawbacks of a vnd approach in Section 2.1. In Section 2.2, we discuss how OpenBSD's vnd(4) driver, FreeBSD's GEOM driver and Linux's loopback mounts do not have flexible key generation mechanisms, do not allow for n-factor authentication and do not adequately protect from dictionary attacks against the pass phrase.

## 2 Architecture and Implementation

In setting out to create a secure local storage facility for Unix workstations, the main design considerations were flexibility, modularity, performance and robustness. The flexibility consideration implied the virtual disk approach. Using the disk-driver interface as the chosen abstraction gives its potential users the freedom to use any kind of file system they want on top of it, be it FFS, swap space, or even space for a database application that writes to the raw partition.

It is important that the design be modular since break-



throughs in cryptanalysis are unpredictable and frequent. Being tied to a single cipher, a single method for choosing IVs or a single method for generating keys would force users to upgrade critical operating system components upon the discovery of weaknesses or vulnerabilities. Modularity also allows for use to be tailored to a specific threat model. Allowing multiple ciphers enables the user to perform cost-benefit analysis based on an evaluation of their risk. Allowing multiple key generation methods allows CGD to be used under different threat models. For example, unattended booting might be a requirement and in this case it would make sense to retrieve the keys from a GSS-API key server. N-factor authentication might be required and CGD provides for this.

The performance consideration led to the decision to place the cryptographic disk functionality below the buffer cache; that is, create a virtual disk driver that directly accesses the raw disk beneath it. This allows all kernel-level (or even user-level) software that makes assumptions about disk layout to work (whether these assumptions are justified is beyond the scope of this paper).

The robustness consideration led to the decision to place the most complex code in user land in the configuration utility. The user land configuration utility performs all key management, key retrieval, *etc.* This leaves the kernel code the easier task of encrypting blocks which makes the kernel code straightforward to audit.

The main disadvantage of the pseudo-disk approach is that there is no per-user keying or any other per-user cryptographic isolation. For better or worse, the security model of Unix is left unchanged. This, of course, is not a problem in the intended user-base for CGD, namely, laptop users or owners of removable storage in national nuclear research laboratories. In any case, if the operating system is not trusted to provide adequate protection, potential intruders can read keys or buffer blocks off */dev/kmem* anyway, so this is not as big a problem as it first appears.

The CGD code base consists of two parts: a kernel driver and user-level configuration software. We examine these two parts next.

## 2.1 The Kernel Driver

Once we decided to create a disk driver, we still had a number of design decisions to make. There are several places in the kernel where we could have implemented such functionality. One quick solution could have been to modify the *vnd(4)* driver. *Vnd* is a device driver that turns a Unix file into a block device. This approach is used in the OpenBSD. There are several reasons why this turns out not to be a good idea. Firstly, in *vnd* requests have to traverse the filesystem code twice. This

increases the number of places where deadlocks can occur. In fact, we have run into such problems when trying to run a system entirely off *vnd* disks on top of *msdosfs* files. Secondly, because we are laying out an FFS (or some other) filesystem on top of a file which has already been laid out in an FFS filesystem, the upper filesystem will make block allocations based on erroneous assumptions of block locality; these destroy many of the advantages of the FFS layout algorithms. Experience with *vnd* actually indicates that this degradation is not that pronounced, but that may be an artifact of the actual workload. Lastly, adding crypto to *vnd* as a way of providing an encrypted disk runs into ease-of-use problems. To do so, a user would have to create a filesystem on the disk, allocate a single maximal-sized file, configure it with *vnd* and partition the resulting pseudo-disk. Attaching and mounting at runtime also add their share of complexity. Complexity for the user is unacceptable unless it delivers value; it is preferable to spend extra effort once if it results in easier use. Moreover, a user will immediately recognize that, for example, */dev/cgd0a* is an encrypted device, rather than have to remember whether they configured the */dev/vnd0a* with or without encryption. This last reason is why we did not release a null encryption transform.

CGD is a pseudo disk, in the same vein as the concatenated disk driver (*ccd(4)*) or RAIDframe (*raid(4)*). *Ccd* takes multiple partitions and presents them as a single disk by either concatenating them or interleaving their blocks (RAID 0). RAIDframe uses the same techniques to provide RAID 0, 1, 4, and 5.

A disk device presents both a block interface and a character interface. CGD does its interesting work in the *strategy()* call in the block interface and the *ioctl()* call in the character interface. The rest of the interface simply presents a normal disk/pseudo-disk in the same way as *ccd(4)*, *raid(4)* or *vnd(4)*. For further information about the block and character devices please refer to "The Design and Implementation of the 4.4BSD Operating System"[11]. For the purposes of this project, we implemented a generic set of functions that could be shared by all of the pseudo-disks to simplify the code.

The *ioctl()* function responds to all the normal disk *ioctl(2)s* and also presents two more: *CGDIOCSET* and *CGDIOCCLR*. The first *ioctl* will cause CGD to attach to an underlying disk device or partition and configure the required parameters (such as the key, the encryption algorithm, the key length, the IV method, *etc.*). The second *ioctl* will detach the CGD from the underlying disk and free the parameters.

CGD's *strategy()* is responsible for scheduling an I/O request. If the CGD is not configured then this call will result in an error. We are passed a *buf* structure which contains all the relevant information about the request



we are about to perform. As with all *strategy()* routines, we perform basic bounds checking and index into the *disklabel* to calculate the real offset of the operation.

If we are reading then we allocate a new *buf* structure and populate it, setting its buffer to be the buffer that we have been passed.

If we are writing then we also allocate a new *buf* structure and populate it, but this time we also allocate memory for the transfer. We do this because the contents of the buffer cache must be stored in plain text and so if we encrypt in place then we would be forced to decrypt when the operation completes. This would double the CPU usage for writing. It is also not clear that this would not cause problems for processes which have *mmap(2)*ed the buffers on which we are working. OpenBSD's *vnd(4)* driver encrypts the data in place and decrypts it when the write completes.

We then pass the newly created *buf* structure to the underlying disk driver. When it completes its work, it will call back into CGD which will check for errors, decrypt the data in the read case and register its completion.

We do not modify the block size of the underlying device because that would violate the atomicity of single writes and the file system code relies on said atomicity to ensure that data can be recovered after a crash. It would also needlessly complicate the driver making it more difficult to audit. We also do not provide integrity checking since this would require the storage of hashes and would break the atomicity assumptions.

We define a modular framework for adding cryptographic algorithms. It is not enough to rely on an underlying framework such as the OpenBSD Crypto Framework because CGD is making additional decisions beyond simply choosing a cipher and a mode. Even with a simple CBC mode the IV needs to be chosen.

In Cipher Block Chaining (CBC) mode, we encrypt each disk block using a block cipher; a different IV is used for each block. We support three ciphers in the initial implementation: AES[6], 3DES[12, 15] and Blowfish[14]. We only support one IV generation method: the block number encrypted under the same key as the data; we do provide the ability in the code to define more if the need arises.

Each block is encrypted separately from any other block. To ward against structural analysis, we use a different IV for each block. In the initial implementation we use as an IV the block number encrypted under the same key as the one used for data. This provides the guarantee that each block has a different IV since the block cipher is a bijection. IVs for successive blocks thus come from plaintexts that differ by very few bits; this may lead to some rather obscure attacks, and we plan to add additional IV generation methods in the next release of CGD.

We decided against trying to use complicated schemes of generating multiple keys for different parts of the disk, as this would increase complexity (which rarely makes for better security) without any identifiable benefit. Moreover, we do not permute the layout of blocks on the disk because it would adversely affect performance, again without any identifiable security benefits.

## 2.2 The Userland Program

Configuration is performed from userland using the *cgd-config(8)* utility. This utility performs all necessary functions including managing the configuration files, generating or fetching keys and configuring the device.

We define two types of configuration file: the main configuration file */etc/cgd/cgd.conf* and per-CGD "parameters files." The main configuration file lists all of the CGD devices that should be automatically configured and the mapping between the CGD and the real disk. The parameters files define per-CGD parameters such as the encryption algorithm, the key generation methods and the IV generation method.

The userland configuration utility needs to derive a key for the encryption. We support an extensible framework for defining mechanisms to derive the key. The parameters file contains a variable number of key generation stanzas. The derived key is the direct sum of the evaluation of all of the stanzas. This provides for *n*-factor authentication.

Currently we support four different key generation methods, namely *pkcs5\_pbkdf2*, *gssapi\_keyserver*, *randomkey*, and *storedkey*.

The *pkcs5\_pbkdf2* method is an implementation of PKCS#5 PBKDF2[2]. If used, it will prompt the user for a passphrase which will then be used to derive the key. PKCS#5 PBKDF2 was chosen because it is well understood and can generate keys of different lengths safely. The use of PKCS#5 PBKDF2 addresses perhaps the most common weakness of otherwise well designed crypto-systems, namely dictionary attacks against the passphrase. Since the method uses chained HMACs and includes an iteration count, it is possible to configure enough iterations to make a dictionary attack arbitrarily prohibitive at a fixed configuration-time cost. The inclusion of a salt from the parameters file precludes an off line attack.

Since it is unlikely that the entropy contained in the pass phrases that users can remember will increase with Moore's Law, it is essential that the default iteration count is chosen with care. The *pkcs5\_pbkdf2* algorithm is run infrequently, so it is acceptable that it takes a reasonably long time to derive a key from the pass phrase. We decided that one second was the appropriate amount of time in light of these considerations. When generating a parameters file, *cgdconfig(8)* will calibrate to the

current hardware and choose an iteration count that will cause the *pkcs5\_pbkdf2* algorithm to take one second to derive the key.

This should provide a reasonable level of protection for the foreseeable future. If we assume that the user's pass phrase contains  $n$  bits of entropy against a dictionary attack, then it will take  $2^n$  seconds to crack. If Moore's Law is that computer speed will double approximately every 18 months, then in  $m$  years it will take  $2^{n - 2m/3}$  seconds to crack. For  $n = 40$ , we get that today it should take  $2^{40}$  seconds = 38865 years. Although the work can be spread over many machines, this is still quite a formidable amount of time. When  $m = 10$ , however, it will take  $2^{40 - 15}$  seconds = 1.06 years which is still a considerable amount of time, but within the capacity of an attacker with sufficient resources.

By contrast, many of the other cryptographic disks use a simple cryptographic hash (such as FreeBSD's GEOM driver) or no transform at all (such as OpenBSD's vnd(4) driver). This provides no protection against dictionary attacks.

The *gssapi\_keyserver* method fetches the key from a keyserver using GSS-API to authenticate and protect the key. This method allows for unattended reboots.

The *randomkey* method reads the appropriate number of bits from */dev/random* and uses that as a key. This method is intended to be used in situations where persistence across reboots is not necessary. Examples would be a swap partition or perhaps a cache that contains sensitive data.

The *storedkey* method has the key in its stanza in the parameters file. It is used for two purposes. First if the parameters file is stored on separate media, such as a USB Mass Storage device, it can be part of an n-factor authentication scheme. Second, we use it to generate new parameters files that have different passphrases but still derive the same key. We shall discuss the latter point later in this section.

In the cases where the user must enter a passphrase, we need a mechanism to detect if the passphrase is entered incorrectly. If verification fails, then *cgdconfig(8)* will print a warning and prompt for the passphrase again. We define methods that use the information that is already present on the disk in its normal usage, such as a disklabel or a filesystem. By using information which is already present we provide no information to an attacker that they could not already intuit from other unencrypted information on the system. We considered storing a hash of the generated key, but this would allow the attacker to perform a dictionary attack against the passphrase even if only in possession of the parameters file. The verification methods that we have defined require that the attacker have both the parameters file and the encrypted disk before a dictionary attack against the passphrase can

commence. The main importance of this is the n-factor authentication where the parameters file is stored on a removable storage device.

We define a framework for the verification methods and currently support three mechanisms: *none*, *disklabel* and *ffs*. The first performs no checking. The *disklabel* and *ffs* methods scan for a disklabel or FFS filesystem after configuration, respectively.

The userland utility also provides the ability to generate additional parameters files which generate the same key. To do this, it fully evaluates the original parameters file including asking for passphrases and retrieving the keys from a keyserver producing key  $K_1$ . It then generates a new parameters file (which may use different key generation methods) and evaluates it producing key  $K_2$ . It then appends a key generation stanza of method *storedkey* where the stored key is  $K_1 \oplus K_2$ . The new parameters file will generate the same key:

$$K_2 \oplus (K_1 \oplus K_2) = K_1$$

The ability to generate new parameters files allows the user to change their passphrase without rekeying the disk. It also allows multiple administrators to access the disk with different passphrases. An administrator could also configure one parameters file with *pkcs5\_pbkdf2* and another with *gssapi\_keyserver* to allow for either a fallback when the keyserver is down, or to allow quicker booting of a laptop in a trusted environment.

The standard NetBSD start up scripts will recognize the existence of the main configuration file and automatically run *cgdconfig(8)* at various points to configure the CGDs. It is necessary to split CGD configuration up in this manner because some of the key generation methods may require network access, such as *gssapi\_keyserver*, whereas some of the disks may need to be configured before the network is brought up. We solve this with the aforementioned tags in the main configuration file.

Tightly integrating CGD support into the default NetBSD start up scripts provides system administrators with the ability to use CGD with a minimum of effort.

### 3 Evaluation

In our current implementation, the kernel code is comprised of the files *cgd.c*, *cgd\_crypto.c*, *cgd\_crypto.h*, *cgdvar.h*, *dksubr.c* and *dkvar.h* located in the directory *src/sys/dev/*. The files *dksubr.c* and *dkvar.c* were written during the course of this project, but are not strictly a part of CGD. They are common functionality which CGD shares with *ccd* and *raid*. Excluding these files, CGD's kernel code is 1348 lines including comments. Of this, the code which interfaces with the ciphers is 91 lines for Blowfish, 103 lines for 3DES and 102 lines for AES. This provides a good metric for how difficult it would be to interface to a new cipher.

The userland program is comprised of all the files in the directory `src/sbin/cgdconfig/`. `Cgdconfig` is 2763 lines of code including comments.

In the following, we analyse the performance of the cryptographic disk. Since the performance varies widely depending on the hardware in question, we analyse three systems: a DEC Personal Workstation 500a (alpha); a Pentium 4-based PC; and an IBM ThinkPad 600E (Pentium II).

Since CGD is designed as a pseudo-disk device, we concentrate our analysis on the raw disk to avoid the secondary effects of the file system code. Our initial speculation was that CGD would increase the latency of a disk transaction by a factor depending on the size of the transfer while using much CPU. The results on the Alpha bear this theory out, while the results on the i386 raise a few questions. We measured disk throughput by reading and writing 100 MBytes from the raw disk devices using different block sizes. The test is a single thread with each operation waiting for the previous operation to complete.

The ciphers had the same relative performance on the different platforms we tested. The only cipher which consistently had a significant detrimental impact on performance was 3DES. The other ciphers performed substantially better.

### 3.1 DEC Personal Workstation 500a

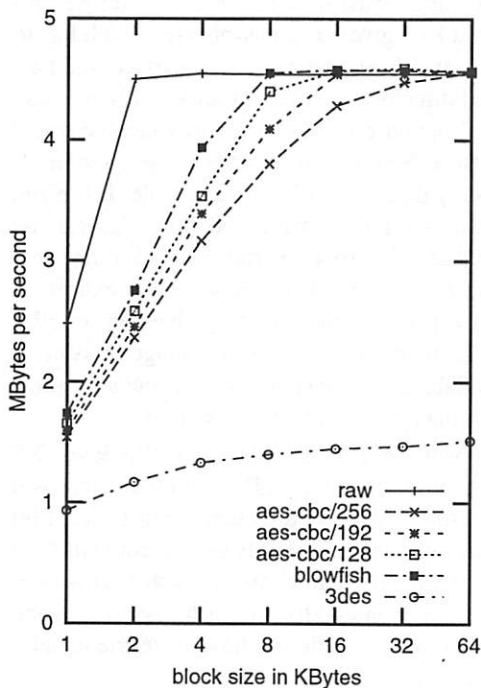


Figure 1: Throughput vs. block sizes of writes to the raw disk devices on PWS500a

These measurements were performed on a DEC Personal Workstation 500a (PWS 500a) running NetBSD

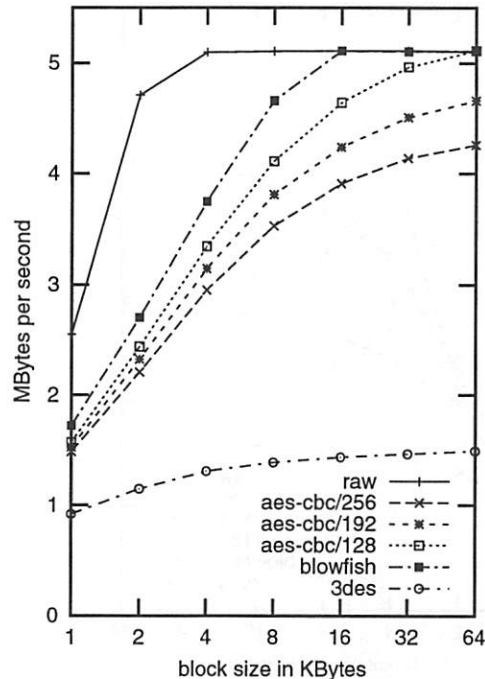


Figure 2: Throughput vs. block sizes of reads from the raw disk devices on PWS500a

1.6I. The PWS 500a has a 500 MHz 21164a Alpha processor. The disk on which the tests were run is a 2GByte Ultra-Wide SCSI disk, a Seagate model ST32155W.

We note in Figures 1 and 2 that at small block sizes performance is degraded quite substantially, whereas as the block size increases performance returns to a quite reasonable level. In the write case AES and Blowfish both quickly began to reach full disk throughput as the block size was increased. The read case proved to be almost as quick, although at the larger key sizes AES could not quite keep up. 3DES performance was poor, as we expected.

NetBSD on the Alpha does not have assembly optimized versions of any of the ciphers used, unlike the i386 architecture.

### 3.2 Pentium 4-based PC

These measurements were performed on a 1.7 GHz Pentium 4 system running NetBSD 1.6Q. The pseudo-disks were constructed on a 19 GByte Ultra100 IDE disk, a WDC WD200BB-00CXA0.

We note from Figures 3 and 4 that results are substantially more complex than they were on the Alpha.

For writing, CGD keeps up at the low block sizes and performance falls off and then starts catching back up. With a 64 KByte block size, Blowfish attains 85% of the raw disk throughput and 128-bit AES attains 73%.

The read performance of the raw disk experiences a large jump between the block sizes of 16KB and 32KB,

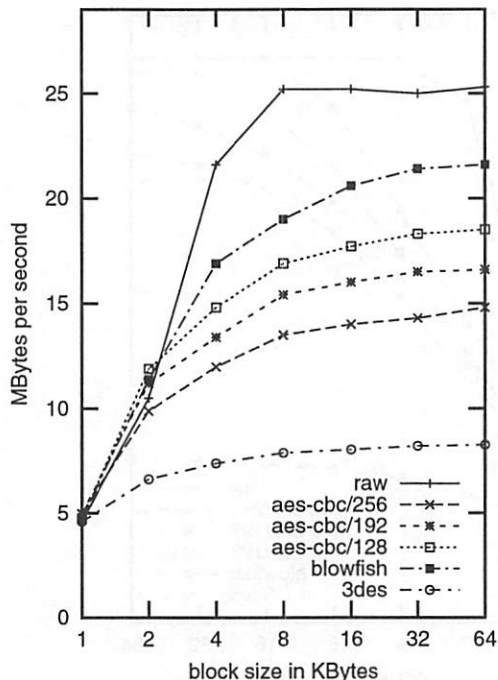


Figure 3: Throughput vs. block sizes of writes to the raw disk devices on P4 class machine

but CGD still maintains the level of performance that one would expect. Blowfish almost keeps up with the raw disk for all of the block sizes and at 64 KByte attains 88% of the raw disk throughput. 128-bit AES attains 64% of the raw disk throughput.

### 3.3 IBM ThinkPad 600E

These measurements were performed on a ThinkPad 600E running NetBSD 1.6Q. The ThinkPad has a Mobile Pentium II processor running at 400 MHz. The pseudo-disks were constructed on a 10 GByte Ultra33 IDE disk, a TravelStar.

The results are much simpler in this case as we note from Figures 5 and 6.

The read and write cases are quite similar. For Blowfish at 64 KByte block sizes, writing achieves 84% of the raw disk throughput and reading achieves 85%. 128-bit AES at 64 KByte block sizes achieves 64% and 58% for writing and reading, respectively.

## 4 Related Work

The Cryptographic File System (CFS)[4] is perhaps the most widely used cryptographic file system. The abstraction it presents is encrypted file contents as well as encrypted file names. It runs as a user-level NFS daemon listening to NFS requests on the loopback interface (so that only requests from the same machine will be honored). This user-level daemon, *cfsd*, provides virtual directories under */crypt*. An encrypted directory hierar-

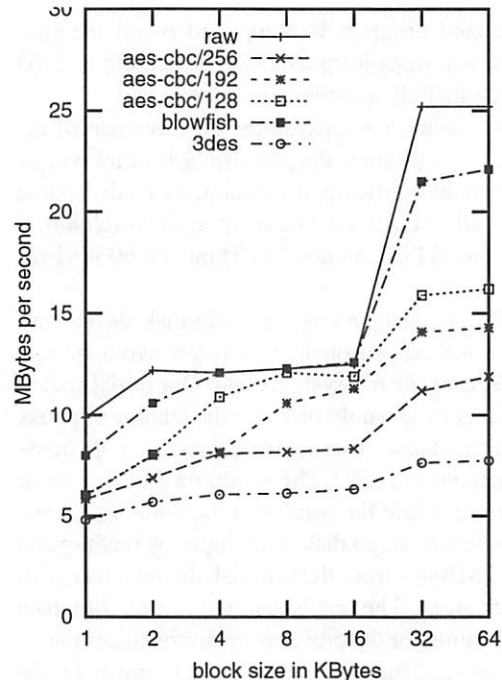


Figure 4: Throughput vs. block sizes of reads from the raw disk devices on P4 class machine

chy to its corresponding cleartext hierarchy, visible under */crypt/username* using the *cattach* utility. Keying is done on a per-user basis; the encrypted hierarchy is initialized with a key, given as a passphrase, which has to be passed to *cattach* when the latter is invoked to attach the encrypted directory. While, of course, only *root* can run the initial *mount* command that provides the */crypt* hierarchy, *cfsd* can be run as an unprivileged server, in which case only that user's files are available. If it is run as *root*, it enforces unix-domain credentials (userid and groupid), so that only processes running under the same userid can access the CFS directories. CFS is extremely useful when a user's home directory lives on an NFS server, and the user wants to take advantage of system-wide file availability and backup services, but still wants more privacy than the NFS primitives allow.

The Microsoft Windows 2000 and Windows XP NTFS file system supports a similar notion of encrypted files and directories. Any subdirectory (or just a leaf file) can be declared encrypted, but only the file contents (and not the file- or directory name) are encrypted. However, the feature is better integrated in the filesystem layout, in that the encrypted files do not have to reside under a special top-level directory.

*Vnd* under OpenBSD supports encryption; each block of the underlying file is just encrypted using a symmetric cipher, as we have already explained in Section 2. There are numerous other examples of loopback disk drivers, such as *ccd(4)* and *vinum*[9], any of which could



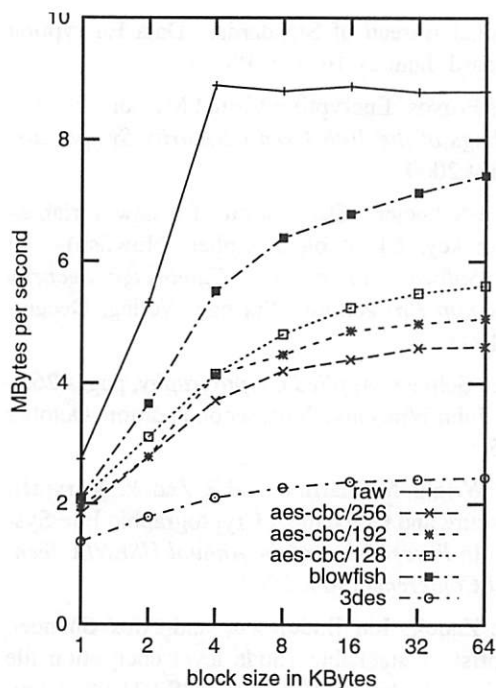


Figure 5: Throughput vs. block sizes of writes to the raw disk devices on ThinkPad 600E

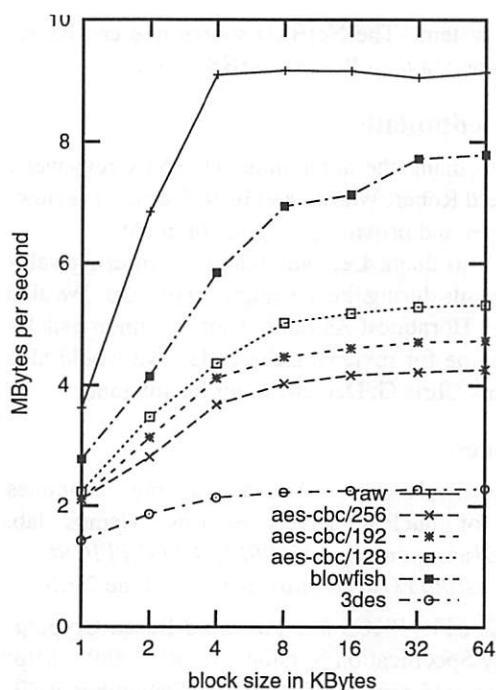


Figure 6: Throughput vs. block sizes of reads from the raw disk devices on ThinkPad 600E

potentially be modified to support encryption, in addition to their primary functionality. Linux loopback devices support encryption either to a file or to a disk partition. BestCrypt[8] is a commercially available loopback encryption device. FreeBSD's GEOM layer supports encrypting a partition. None of these use PKCS#5 PBKDF2 or any other iterated salted key generation method and so do not offer adequate protection from dictionary attacks against the pass phrase.

Another driver that closely resembles CGD is the encrypting swap device[13] on OpenBSD. In order to protect against keys or other sensitive information lingering in the swap space long after their corresponding processes have terminated, the swap device itself is encrypted. Their keys are chosen randomly, and change frequently, thereby invalidating old data.

Half-way between the loopback drivers and the NFS approaches lie encrypting file systems implemented as a vnode layer. Stackable vnodes were introduced by Rosenthal [7], and were used to implement among other things two encrypting file systems Cryptfs [17, 18] and NCryptfs [16]. This approach offers substantially higher performance than the NFS approaches, but is less portable.

## 5 Conclusions and Future Work

Many computers fall into the category of "trusted administrator, low physical security" and for these computers an encrypted disk provides adequate protection against

typical threats. We have designed and implemented an encrypted disk, taking care to avoid some of the most common errors that have been made in other systems by ensuring that it is easy to configure and the cryptography is designed according to best practices.

We have demonstrated that the performance of our cryptographic disk is acceptable, although the exact performance is quite dependent on the hardware configuration. Blowfish is quite fast, suffering only about a 15% performance hit on the platforms we tested and AES offers performance that is still acceptable.

CGD is in use today, both in the laptop of one of the authors and in the laptops of many NetBSD users. One user even reported that his use of CGD has already kept the contents of his laptop from the prying eyes of one of his client's employees who decided to use a Linux boot floppy to examine his laptop while he was taking his lunch break.

Because CGD defines extensible frameworks for many aspects of its configuration, there is much room for future growth. Currently CGD supports three ciphers and one IV generation method. We may add additional ciphers if there is demand. We plan on adding additional IV generation methods.

There is much room for future work on CGD. Additional key generation methods could be defined. CGD could be modified to use the *crypto(9)* API which was not available in NetBSD when CGD was written.

The software is freely available as part of the NetBSD

operating system. The NetBSD source tree can be accessed online via <http://www.NetBSD.org/>.

## Acknowledgments

We wish to thank the anonymous USENIX reviewers, our shepherd Robert Watson and Erez Zadok for reviewing the paper and providing helpful comments.

We wish to thank Lee Nussbaum for offering valuable comments during the development of CGD. We also thank Love Hörnquest Åstrand, Thor L. Simon and Jason R. Thorpe for reviewing the code. We would also like to thank Chris G. Demetriou for his tolerance.

## References

- [1] Amy Paulson. Senate hearing examines loss of nuclear secrets at Los Alamos lab. <http://www.cnn.com/2000/ALLPOLITICS/stories/06/14/losalamos.hearing/>, June 2000.
- [2] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898, <http://www.ietf.org/rfc/rfc2898.txt>, September 2000.
- [3] Betsy Schiffman. Stolen Qualcomm Laptop Contains Sensitive Data. <http://www.forbes.com/2000/09/19/mu5.html>, September 2000.
- [4] M. Blaze. A Cryptographic File System for Unix. In *Proc. of the 1st ACM Conference on Computer and Communications Security*, November 1993.
- [5] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In *Proceedings of the Annual USENIX Technical Conference*, June 2001.
- [6] Joaen Daemen and Vincent Rijmen. AES Proposal: Rijndael. <http://www.esat.kuleuven.ac.be/~rijmen/rijndael/>, June 1998.
- [7] David S.H. Rosenthal. Evolving the Vnode Interface. In *Proceedings of the Annual USENIX Technical Conference*, June 1999.
- [8] Jetico, Inc. BestCrypt software home page. <http://www.jetico.com/>, 2002.
- [9] Greg Lehey. The Vinum Volume Manager. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, June 1999.
- [10] John Leyden. For sale: memory stick plus cancer patient records. <http://www.theregister.co.uk/content/55/29752.html>, March 2003.
- [11] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*, pages 196–204. Addison-Wesley Publishing Company, 1996.
- [12] National Bureau of Standards. Data Encryption Standard, January 1977. FIPS-46.
- [13] Niels Provos. Encrypting Virtual Memory. In *Proceedings of the 10th Usenix Security Symposium*, August 2000.
- [14] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*. Springer-Verlag, December 1993.
- [15] Bruce Schnier. *Applied Cryptography*, pages 265–301. John Wiley and Sons, second edition, October 1995.
- [16] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, June 2003.
- [17] Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998.
- [18] Erez Zadok, Ion Badulescu, and Alex Shender. Extending File Systems Using Stackable Templates. In *Proceedings of the Annual USENIX Technical Conference*, June 1999.

# xstroke: Full-screen Gesture Recognition for X

Carl D. Worth  
*Information Sciences Institute*  
*University of Southern California*  
Arlington, VA, 22203  
cworth@isi.edu

## Abstract

Gesture recognition is a common method of text input on handheld and other pen-based computing devices. Xstroke is a full-screen gesture recognition program for the X Window System.

The touchscreen of a typical pen-based device is divided into two regions, a primary region for application display and interaction, and a secondary region for gesture input. Full-screen gesture recognition improves on the typical implementation by sharing the entire screen for both purposes. Applications benefit as more screen real estate is available. Recognition performance improves due to increased information from larger gestures. Usability increases as less time is lost switching attention between two separate screen regions.

Full-screen gesture recognition presents several user-interface design challenges. Conventional GUI interaction is pointer driven. This makes it difficult for systems with single-button pointer devices to distinguish between GUI interaction and character input. Several solutions to this problem are examined.

The simple feature-based recognition engine at the heart of Xstroke has proven capable of resolving gesture sets of 100 different gestures with up to 95% accuracy. Xstroke is freely available and has become the predominant gesture recognition system for X-based handheld devices.

## 1 Introduction

Gestures, or marks entered with a stylus or mouse to invoke commands, have become an increasingly relevant aspect of user interfaces over the past few years. Gestures are an efficient means of command input since a gesture can simultaneously indicate both the operator and operand for a command. On the desktop, gestures have long been present in specialized fields such as CAD, but have recently begun to appear in general-purpose applications including text editors [3] and web browsers [13, 8].

Outside the desktop arena, gestures are perhaps more compelling since many handheld and other mobile computing devices provide a stylus as a primary input device. With these devices, software is required to allow the user to efficiently input text using the stylus. Gestural text input is possible if the gesture recognition system supports

a gesture set large enough to cover every desired character. It is also desirable if the recognition system is capable of recognizing gestures that are similar to character shapes in their natural forms.

Xstroke is a gesture recognition system designed to add gestures to the desktop and to provide efficient character input for pen-based computers running the X Window System. The X Window System is the standard graphical user for UNIX and UNIX-like operating systems. The XFree86 project [15] distributes an implementation of the X Window System including a tiny X server known as kdrive [14] which is suitable for use on handheld computers.

xstroke has several distinguishing features:

**Full-screen recognition** xstroke allows gestures to be entered anywhere on the screen, directly on top of the program to receive the gesture command. The current stroke is drawn in translucent “ink” above other programs.

**Extensible recognition** xstroke includes a simple feature-based recognition engine, and can be extended with more sophisticated recognition engines through dynamically loaded libraries.

**Adaptive slant correction** xstroke automatically adapts its recognition to account for changes in the slant or rotation of input gestures.

**Configurable alphabet** xstroke includes a default gesture set designed to allow efficient use of xstroke as a keyboard replacement. The gesture→action mapping can be customized and custom gestures may be added to the gesture set.

### 1.1 Related Work

Gesture recognition can be added to user interface environments at several different layers including direct implementation within an application, recognition within a library or toolkit, or recognition through an external program. Each strategy has different implications on the effort needed to integrate the recognition support into applications.

Open source/free software programs directly supporting gestures include emacs/xemacs [3] and mozilla [8]. In both of these programs, the recognition engine is included as an integrated part of the application and is not available to other programs.

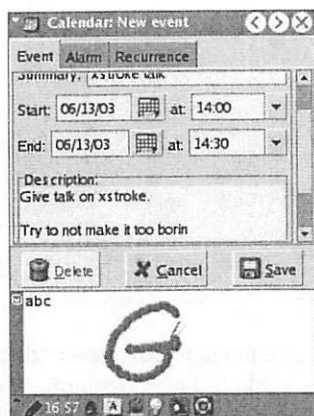


Figure 1: xstroke in window mode

If the recognition engine is placed within a library, it can be used by multiple applications. James Kempf developed a generic library interface [11] for handwriting recognition engines, which has not achieved widespread use. Mark Willey's recognizer, libstroke [16], is an example of a specific recognizer implemented as a library. Incorporating gesture support at this level requires modifying the application to pass gestures to the recognition library and act on the recognized results. libstroke has been used in modified versions of ggv, gEDA, and the FVMW window manager.

Several efforts at providing system-level gesture support implement recognition within the user-interface toolkit. This is the approach taken in SATIN [10] with a Java toolkit; Artkit [9], a custom toolkit supporting gestures; and in a modified version of the Garnet toolkit [12]. The benefit of toolkit integration is that the gestures become available to all applications using the toolkit, without any gesture-specific code required in the application. Naturally, applications designed without using the toolkit do not get the gesture support. Neither of the popular modern open source/free software toolkits, (KDE [5] and GNOME [6]), offer gesture support.

The final approach for integrating gesture recognition with the user interface is as a peer application. The benefit of this approach is that gesture support can be provided to applications without modification to those programs. Existing programs using this approach include xscribble and wayV [4].

The author's experience, (and frustration), with xscribble led directly to this work. Recognition performance was poor and there were no readily available documentation on how to re-train the recognizer. These frustrations led directly to the fact that all of the gesture definitions for xstroke are in human readable form and can be modified online with no need for external tools beyond the ability to edit a text file.

The author was unaware of wayV until after xstroke had been written. Like xstroke, wayV is a full-screen gesture recognition program for the X Window System.

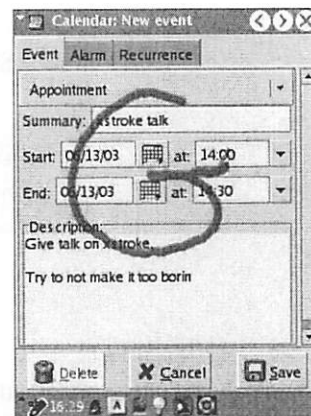


Figure 2: xstroke in full-screen mode

Gesture recognition is distinguished from conventional pointer interaction by the use of a modifier key or a specific pointer button. Configurable gesture actions include program execution and generation of keypress events.

## 2 Full-screen Recognition

Initial versions of xstroke required gestures to be entered into a window belonging to xstroke. Figure 1 shows xstroke being used in this way to enter text into a calendar application. Entering gestures into a separate window has several drawbacks:

- On small displays, the recognition window steals precious screen real estate from the active application. Typical handheld computers that run xstroke have a display size of  $320 \times 240$  pixels. Application developers are hard-pressed to get all the information they would like into those pixels without one fourth of them being used for input and unavailable for output.
- Restricting gesture recognition to a window places a maximize size on the gestures that can be input. Smaller gestures make it harder for the recognition engine to extract the desired features.
- Gestures in a separate window cannot achieve their full expressive power. A gesture in a window can indicate an operation by its shape, but cannot indicate the operand for the operation by its position.

Current versions of xstroke allow gestures to be entered anywhere on the screen as shown in Figure 2. This has benefits that are the opposite of each of the drawbacks listed above: the entire display is available for active applications, gestures may be entered as large as desired leading to more accurate recognition, and the position of gestures may be used to indicate operands for gesture commands.



## 2.1 Resolving Pointer Ambiguity

Full-screen gesture input is a compelling feature, but it also provides some significant challenges in the user interface design. The main problem is that full-screen gesture recognition introduces an ambiguity in the handling of pointer motion events. Namely, does a sequence of pointer motion events indicate a gesture for recognition, or is intended for direct manipulation of graphical elements of the user interface, (ie. standard mouse interaction)? xstroke implements several complementary mechanisms for resolving this ambiguity.

### 2.1.1 Reserved Button

If multiple pointer buttons are available, xstroke can be configured to perform recognition only when a specific button is pressed. If a button can be dedicated to xstroke this approach is sufficient and can be quite satisfactory. However, for many handheld systems, the only pointer device is a simple touchscreen which acts as a single-button pointer device. In that case, other mechanisms are necessary for managing when recognition occurs.

### 2.1.2 Manual Toggling

xstroke provides a small control window with a button that can be used to toggle recognition. When recognition is enabled, xstroke intercepts all pointer motion between button-down and button-up and performs recognition on the resulting gesture. When recognition is disabled, all pointer events are handled as if xstroke was not present. This toggle capability provides the necessary mechanism to allow the user to control the handling of pointer events, but it is far from convenient. Moving the pointer back and forth from the active application where gestures are being performed to the control window is wasted user time. This is especially costly when the display is large, (causing the control window to be far from the active application), or when toggling recognition is frequent, (such as the need to continually scroll a document while writing large amounts of text).

### 2.1.3 Automatic Toggling

The author regards manual toggling of recognition as a feature of last resort. Gestures are powerful enough that full-screen recognition should always be enabled. Instead of requiring the user to turn recognition off to use a button or a scrollbar, the system should “know” that recognition is not desired on those elements and should automatically disable recognition when appropriate.

For the case of buttons, a simple solution proves quite effective. In the default configuration, xstroke recognizes a single pen tap as a gesture associated with an action to synthesize a button press event. Because of this, GUI buttons can be used directly without having to disable recognition.

The only remaining problem then is how to automatically disable recognition for GUI elements that require

pointer motion while the button is pressed, (ie. click-and-drag events), for example, a scrollbar. The problem was solved for buttons by recognizing a gesture, then synthesizing pointer events. That approach does not work here since the gesture is not recognized until the pen is lifted, while the user will expect the scrollbar to respond while the button remains pressed.

We experimented with automatically disabling recognition based on a heuristic for determining if the window under the stroke “expected” recognition or not. The heuristic is to examine the event mask of the window at which the gesture begins to see if it has selected for KeyPress events. The rationale is that most gestures are used to send synthetic keypress events, so if a window has not selected for keypress events, then recognition can be disabled and pointer events can be passed through directly. This idea had some success. With xterm, xstroke could be used for recognition within the terminal window and for operating the scrollbar. With GTK+ applications, scrollbars and menus could be used without manually disabling recognition.

But, at the same time, the complex window hierarchies within some application cause this heuristic to fail, sometimes with disastrous results. For example, with Qt applications, the scrollbars and menus are still not usable without manually disabling recognition. This failure mode is acceptable.

However, with the window hierarchy of rxvt, for example, the child window does not select for KeyPress events, (instead a parent window does). In this case the xstroke heuristic disables recognition for the entire window, and xstroke cannot be used to input text into rxvt. A similar problem happens with some GTK+ dialog boxes with multiple text boxes. Recognition is disabled for the majority of the dialog box, except for strokes that actually began inside one of the text boxes.

These failure modes are so bad, and the heuristic fails often enough that it was decided that attempting context-sensitive enabling/disabling of stroke recognition based on heuristics is infeasible.

Context-sensitive recognition would still be very powerful—if it could be done reliably. One possibility is that a convention could be developed so that windows could explicitly state whether recognition should be enabled on each window or not. It is not anticipated that many applications will be modified to provide these hints. Rather, it is expected that toolkits will provide these hints for those classes of windows which require them, (such as scrollbars). A more general solution that would also be very desirable is if toolkits would provide some indication of the widget-class of each window, (eg. menu, button, scrollbar). Xt implements a feature like this with the `_MIT_OBJ_CLASS` name, but most modern toolkits provide no such indication.

With a few carefully placed (and standardized) hints in place within the toolkits, the pen should just do the

right thing—allowing full-screen recognition, but also enabling scrollbars, menus, and other GUI widgets to still be manipulated.

### 2.1.4 Tap-and-hold

Finally, there is one more class of window that still needs to be discussed. Some windows want recognition events, but also act on click-and-drag pointer events. The most common example is a text widget which could use gesture recognition for text input, but also allows click-and-drag to select text. A simple hint controlling whether recognition should occur or not is not sufficient here. For this case, xstroke has another toggle mechanism that is considerably faster than the global toggle button in the control window. If a gesture begins with a tap-and-hold motion, (that is, if there is little or no pointer motion within a short timeout period following a button-down event), then xstroke will disable stroke recognition for a single gesture.

With this mechanism in place, the text widget becomes quite usable. In the most frequent usage, gestures are used on the text widget to input text. Then, when an occasional text selection is required, a quick tap-and-hold, (by default .75 seconds and naturally configurable), is all that is needed to make the pointer select text rather than recognize a gesture. xstroke also changes the pointer cursor, (from a pencil for recognition to the text selection bar), to indicate that the timeout has expired and the user can begin the selection operation.

## 3 Recognition

Within xstroke, gestures are recognized with a simple feature-based engine. An input gesture is captured as a sequence of time-stamped pen positions. It is then converted into a sequence of discrete feature values with the goal of selecting the correct target gesture.

The gesture feature vector is then compared with each configured gesture class. When the input gesture matches a gesture class exactly for each feature, that gesture class is returned as the recognized class.

The currently implemented feature recognizers are quite limited. But, the recognizer interface is well-defined so that it can easily be extended. New recognizers can be added as dynamically loaded modules for ease in experimenting. The following sections describe in detail the feature recognizers currently available in xstroke.

### 3.1 Grid Recognizer

The primary feature implemented in xstroke is the grid feature. This feature was inspired by the algorithm used in libstroke[16] though xstroke features a custom implementation with some improvements. The grid feature is based on a  $3 \times 3$  grid numbered from 1 to 9. The grid is centered on the gesture and is generally scaled independently in the X and Y dimensions to fit the bounding box of the gesture as shown in Figure 3.

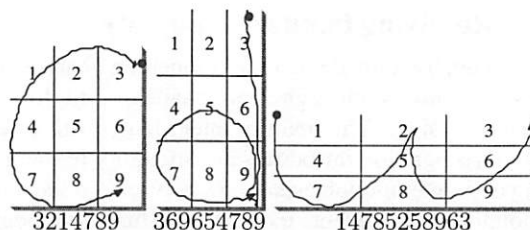


Figure 3: Grid fit to 3 different gestures

The automatic fitting of the grid allows for consistent recognition in spite of gesture variation in scale or in stretching in one dimension or the other. However, for some strokes that are very wide or very tall, the deformation introduced by scaling the grid will have a negative impact on recognition. For example, this grid scaling would introduce an ambiguity between a gesture consisting of a  $45^\circ$  line and a gesture of a very nearly horizontal line. To avoid these errors, the following rule is applied. Whenever the bounding box of a gesture is more than 4 times larger in one dimension than the other, the grid is fit to the smallest square containing the gesture. This is demonstrated in Figure 4.

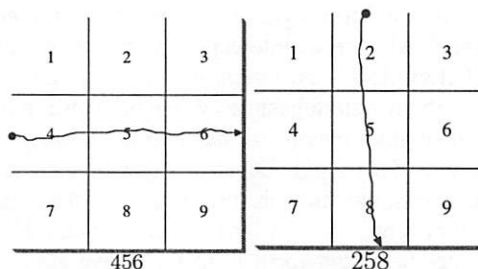


Figure 4: Grids constrained to be square for wide and tall gestures

Once the grid has been fit to the gesture, the feature value is determined by simply traversing the gesture points in time order and recording the number of each grid cell as the gesture passes through it. For example, the horizontal line in Figure 4 has a value of 456 while the 'C'-shaped stroke of Figure 3 has a value of 3214789.

There are two details in the extraction of the grid cell sequence worth noting. First, depending on the sampling rate of the input device and the speed of the gesture input, there may be gaps between samples large enough to cause a cell to be missed. To avoid this problem xstroke uses linear interpolation between every sample point, (using a Bresenham algorithm), and examines the cell of every interpolated point.

Second, the gesture may pass through a cell so briefly that perhaps that cell value should be considered noise and discarded. Libstroke [16] attempts to solve this problem by disregarding cells which are passed through

by some small percentage of the total number of sample points. Early experiments using libstroke within xstroke demonstrated two problems with this approach. First, discarding cells can result in an “impossible” digit sequence with the gesture jumping from one cell to a non-adjacent cell. Second, as gestures become more complex, the static threshold results in a greater number of cells being discarded. With extremely long gestures, an empty digit sequence may be generated.

In order to avoid discarding useful information describing the shape of the gesture, xstroke does not attempt to filter the grid digit sequence. As a result xstroke must be able to match sequences with more noise, (and hopefully more information), than those provided by libstroke. The sequence matching technique used in xstroke is described below.

### 3.2 Regex Sequence Matching

After generating a digit sequence for a gesture, libstroke, (as well as early versions of xstroke), uses exact string comparison to match the sequence with configured gestures. This approach can be problematic. Consider a gesture that is a diagonal line from the upper-right to the lower-left, (this is the default gesture in xstroke for the Return key). The ideal digit sequence for this gesture would be 357, but since the ideal stroke passes near cell intersections in the grid, significant variation in the digit sequence can result in practice. Figure 5 demonstrates two of the many possible digit sequences that might result.

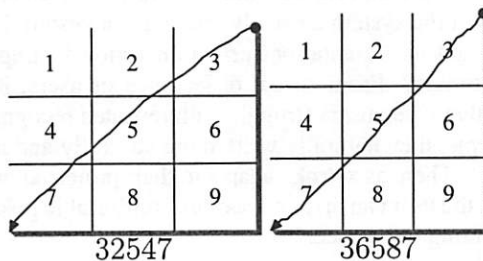


Figure 5: Similar strokes with different digit sequences

In order to reliably recognize this gesture with exact string matching, the recognizer would have to be configured with all likely variations of the digit sequence: 357, 3257, 3657, 3547, 3587, 32547, 32587, 36547, 36587.

For this very simple gesture class, it is necessary to store 9 separate strings in the recognizer! This gets tedious very quickly, especially since an increase in stroke complexity yields an exponential increase in the number of possible sequences, (generally  $3^N$  possible sequences for a stroke passing near  $N$  cell intersections). Clearly, this is not a scalable approach.

The solution in xstroke is that the grid feature value for a class of gestures is represented by a regular expres-

sion. For the case above, the default xstroke configuration contains the following specification: `Return = grid("3[26]?5[48]?7")` This regular expression provides an efficient way to store the class of 9 different sequences. The matching of a gesture sequence with a class is also efficient as regular expression matching is generally provided by a highly optimized system library.

This same technique can be used to encode robust gesture classes for sophisticated shapes. As a rather extreme example, the following regular expression represents a robust class of ‘B’ shapes for xstroke: `b=grid("( [12]*[45][78] | [12][45]+[78]? )? [78]*[4]*(1?[2][369]+ | 1[25][369]* ) ([369]+[25]+8?[147]?[258]*[369]+ | [25]*8?[147]+[258]+[369]* ) ([369]*[58][74]+ | [369]+[58][74]* ) ")`. This single expression describes the three ideal variations of the ‘B’ shape shown in Figure 6 along with the myriad sequences that may result when these gestures are input.

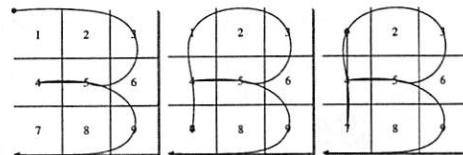


Figure 6: Three ideal ‘B’ shapes

The expression above can be daunting to read, (let alone attempt to understand). Its existence demonstrates that xstroke did not achieve all of its original goals — it had been hoped that the xstroke configuration file would always be easy to read and edit.

But the complex ‘B’-shape expression also proves an interesting result. The author originally chose the grid recognizer since it would be so easy to implement. It was never expected that it would be able to handle anything beyond the most simple gestures. But, with expressions such as the one above, this simple recognizer is capable of distinguishing some rather sophisticated gestures. The grid recognizer has been so successful that investigation into other recognizers has proceeded slowly since the grid recognizer is often good enough.

### 3.3 Anchor Recognizer

xstroke contains a special-purpose recognizer known as the anchor recognizer. This recognizer cannot distinguish much of the shape of a gesture, but is used to “anchor” a gesture in absolute position and scale. For example, this allows xstroke to distinguish a gesture made in the middle of a touchscreen from a gesture of a similar shape that begins and ends at the screen edges. The anchor recognizer is based on the grid recognizer but has two important differences.

First, for the anchor recognizer the  $3 \times 3$  grid has cells of non-uniform size. The center cell is enlarged to fill



most of the grid area (roughly 80 as very small rectangles positioned in each corner. The second difference is that the anchor grid is not scaled to fit the bounding box of the gesture but is scaled to cover the entire gesture area.

Figure 7 shows two gestures of similar shape; the grid recognizer returns the same digit sequence for each gesture. However the absolute position of the gestures is different and this is distinguished by the anchor recognizer. In practice, users of xstroke have found these global edge-to-edge gestures useful for strokes useful for invoking various operations. For example, a gesture from the bottom of the screen to the top might launch a program to display the currently configured xstroke gesture, or a diagonal slash from one corner of the screen to the other might close the current application. Using anchor, the gesture shapes for each letter in the alphabet can be reused for new operations besides their standard use for character input. For example, an edge-to-edge 'e' shape might be used to launch an email program.

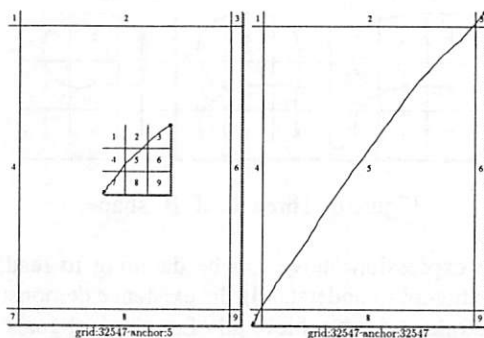


Figure 7: Anchor recognizer distinguishes absolute position and scale

### 3.4 Adaptive Slant Correction

There is a significant amount of variation in gesture orientation from one user to the next. In addition to the natural differences of slant due to different writing styles, there is added slant when xstroke is used on a handheld computer. Typically, one hand holds the computer while the other hand holds the pen. Due to the physical arrangement of human shoulders and arms, the most natural way to hold the computer in this way does not lend itself to precisely upright character entry. [XXX: It might be interesting to measure the natural slant in various users simply holding the device in a comfortable way].

Uncorrected slant leads to misrecognition as many desirable strokes differ only in orientation. Initial testing of xstroke without slant correction showed some users unable to achieve acceptable recognition rates. The need for slant correction has been shown in various other character recognition systems as well [7].

To correct for this, xstroke allows an OrientationCorrection action to be associated with any gesture. The

OrientationCorrection action accepts a numeric argument giving the ideal orientation of the gesture that was recognized. For example, the left-to-right gesture used for Space in the default alphabet has an OrientationCorrection of 0 degrees while the right-to-left gesture used for BackSpace has an OrientationCorrection of 180 degrees. All of the simple one-line strokes have similar OrientationCorrection actions associated with them.

Whenever xstroke correctly recognizes a gesture with an OrientationCorrection action, it measures the actual angle from the first point of the stroke to the final point. It subtracts from this angle the ideal angle giving in the OrientationCorrection action to compute the expected slant at which the user is holding the device. xstroke uses an average of the last 5 values computed in this way as an estimate for the current gesture orientation. Then, as each gesture is performed, before the gesture is passed to the recognition engine, it is rotated by the current orientation estimate to compensate for any slant.

This system has proved quite effective at helping to alleviate problems with recognition from holding the device at various angles. This system gets great benefit from the fact that the Space gesture, the most common character in text, provides orientation information. Also, in the presence of recognition errors due to misestimated orientation, the Backspace gesture will often be performed to correct the errors, and will provide the needed information for orientation correction as well.

It may seem strange that the orientation correction actions only occur after characters are correctly recognized. If the current orientation estimate is incorrect, how will the system correctly recognize a gesture in order to get an OrientationCorrection action to improve the estimate? From casual observance of users, it appears that when users struggle with repeated recognition problems, they naturally write more carefully and more upright. Then, as xstroke adapts to their preferred orientation, the user can drift into a more comfortable position for holding the device.

## 4 Customization

Like many successful open systems, xstroke is designed to be modified and extended. The goal has been to design a system with sane defaults in which all policy decisions can be customized by the user. Several aspects of the user-interface can be configured such as which button triggers recognition and whether recognition is limited to a single window or is available full-screen. xstroke also allows the customization of the gesture set and extension of the set of feature recognizers. These aspects are discussed in detail below.

### 4.1 Default Gesture Set

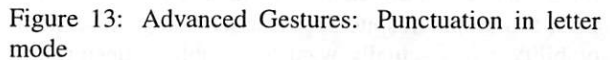
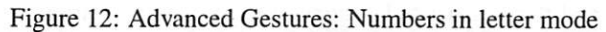
The current release of xstroke contains pre-defined gestures for all printable ASCII characters, (eg. letters a-z, digits 0-9, and punctuation). It also includes addi-



The basic gesture set does include several variations for many letters. These are intended to accommodate multiple writing styles and to improve efficiency. For example, the vertical bar on many letter shapes, (eg. B, D, R), is optional which allows faster gesture entry.



Cramming every key on the keyboard into a single gesture set requires some creativity to avoid several



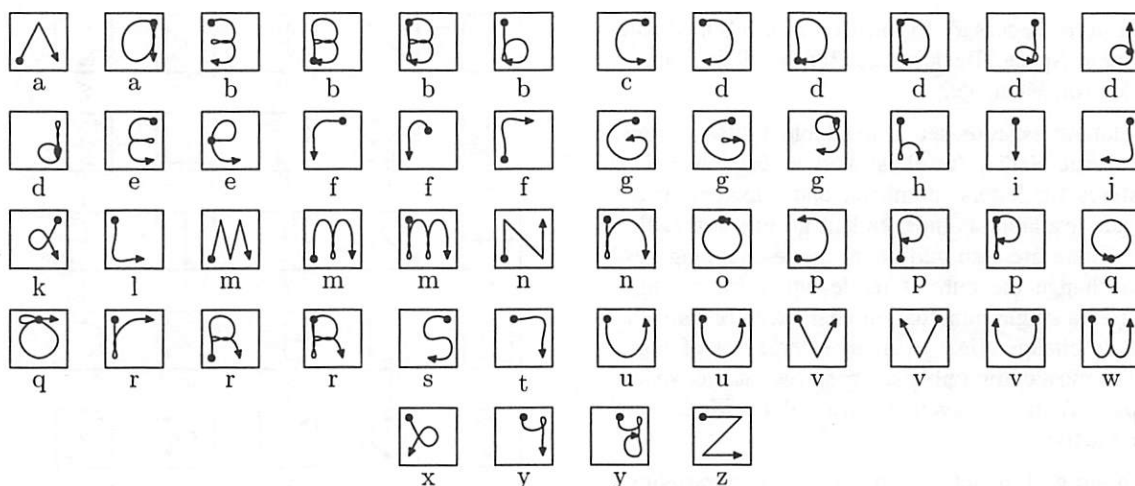


Figure 9: Basic Gestures: Letters

clashes of similar shapes (eg. ‘O’ vs. ‘0’, ‘S’ vs. ‘5’, etc.). The “advanced” gesture set solves these problems by including gestures for each number that are drawn in the opposite direction, (eg. the gesture for the letter ‘O’ is drawn in a counter-clockwise direction while that for the number ‘0’ is drawn clockwise, ‘S’ is drawn from top to bottom while ‘5’ is drawn from bottom to top, etc.).

The advanced gesture set is a strict superset of the basic set, so the advanced set is enabled by default in the current system. The only reason to distinguish between the two sets is that the documentation for the basic set is simpler and more suitable for new users.

## 4.2 Customizing Gestures

The gesture set for xstroke is contained in a plaintext configuration file to make it easy to customize gestures and their associated actions. One common customization desired by many users is the ability to input accented characters. A simple approach is to change the gestures for standard punctuation symbols into their “dead key” equivalents. This enables multi-stroke combinations to be used to input accented characters in the exact same manner as on a keyboard.

There is a significant potential problem caused by allowing the end-user to customize the actual gesture shapes, (as opposed to the gesture actions). Namely, good gesture set design is a difficult task.

One aspect of the difficulty of gesture design is that visual similarity of gestures can make it hard for users to remember gestures correctly [1]. The author does not claim to be an expert in gesture set design nor does he claim understanding of human perception of similarity. It is hoped that exposing customizable gestures will allow users to choose gestures that are easy to remember for themselves individually. In fact, a survey of PDA users found that in spite of problems with gesture memorability, users actually want to be able to design new

gestures [2].

A second difficulty in gesture set design is the need to understand the gesture recognition engine in order to be able to determine if two different strokes can be reliably distinguished. An intentional design decision of the feature recognizers within xstroke can help with this problem. For all xstroke recognizers, the features were designed so that feature values would be meaningful and useful to humans.

This allows the possibility for the recognition system to reason about ambiguities in the gesture set and report them in a way that is meaningful to the user. This has already proven useful in improving several misclassification problems in xstroke, (such as the difficult distinction between ‘v’ and ‘u’).

## 4.3 Extending the Recognizer

The emphasis on human involvement in the training process is not meant to suggest that automated techniques should not be used. It would be very convenient for users to be able to specify new gestures by example, and xstroke could benefit greatly from machine learning techniques to adapt to the user. However, limitations may exist within a feature set that no amount of machine learning can overcome. This is one reason xstroke has a well-defined mechanism for loading additional feature recognizers through dynamic libraries.

As a concrete example, consider Figure 14 which shows two ‘h’-shaped gesture and an ‘L’-shaped gesture. The second ‘h’ is misclassified as an ‘L’ since the hump is not larger than 1/3 the height of the gesture. This is one of the most common current misclassification problems experienced by the author with xstroke.

This misclassification could not be fixed through re-training. If a user retrained the misclassified ‘h’ as an ‘L’ all subsequent ‘L’-shapes would be misclassified as ‘h’. This would not be obvious to a user dealing exclu-

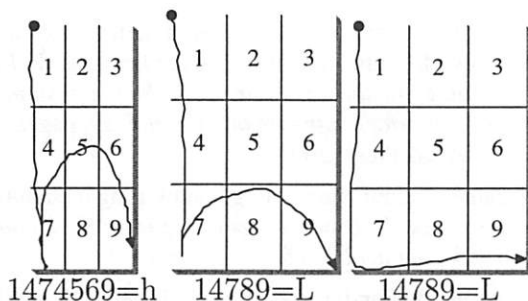


Figure 14: Misclassification of 'h' as 'L'

sively at the level of gestures. But when looking at the grid feature values the problem is immediately apparent, and it is obvious that no quick fix is available since two distinct shapes both have a feature value of 14789.

The only solution to this problem is to introduce a new feature which could distinguish between these shapes. xstroke currently has an experimental direction recognizer intended to address this problem. Just as the grid recognizer creates a short string based on quantizing 2D position into 9 values, the direction recognizer creates a string based on quantizing the stroke direction at each point into 8 directions. With the addition of this recognizer, the 'h' and 'L' strokes could then be easily distinguished:

$$\begin{aligned} h &= \text{grid}(14789) \cap \text{direction}(\downarrow \nearrow \rightarrow \searrow) \\ L &= \text{grid}(14789) \cap \text{direction}(\downarrow \rightarrow) \end{aligned}$$

## 5 Evaluation

Evaluating a new recognition system such as xstroke is a difficult task. It would be ideal to have access to a large database of multi-user gesture input data. Some large databases of gesture and handwriting samples are available commercially, but the data could not be redistributed, so those databases are not feasible for an open source project such as xstroke. It would be an interesting project to assemble an open corpus of stroke data, (which is especially feasible since in recent years there has been a great increase in the number of users of open source software that have access to touchscreen devices).

The custom gesture set in xstroke also complicates testing since it may be difficult to separate measurements of the performance of the recognizer from measurements of the level of expertise that the users have with the custom gesture set.

The author must certainly be considered an expert user having designed the gesture set. The author's own gestures were also the primary source of data for the manual training of the recognizers. The author typically achieves successful recognition of 95% of gestures input. Of the unsuccessfully recognized gestures roughly half are misclassified and half are rejected.

xstroke has supplanted xscribble as the standard character recognition program in the X11-based distributions

from handhelds.org. There is anecdotal evidence to suggest that some users find xstroke to have more reliable recognition than character recognition software on popular commercial PDAs.

This paper has introduced several issues such as the user-interface issues of Section 2 and the advanced gesture set of Section 4.1. It would be very interesting to see a usability study investigating these issues, but such a study is beyond the scope of the current work.

## 6 Future Work

As discussed in Section 1.1 there are several approaches that can be used for integrating gesture recognition into a user-interface. The current mechanism of xstroke as an independent application requires the least modification to other programs, but also fails to provide tight integration with applications. It would be extremely useful to preserve the capabilities of the current xstroke interface but to split the actual recognition engine into a library that could also be used by other applications and toolkits.

Tighter integration with other applications would allow those applications to act on aspects of the gesture itself, rather than just the synthetic event triggered by recognition. For example, a scribble gesture might be used to indicate the deletion of the text covered by the gesture.

A significant limitation of the current user-interface is that only single-stroke gestures are collected for recognition. There is no inherent limitation in the recognition engine that would prevent it from recognizing multi-stroke gestures. This capability would allow much more natural letter forms to be used.

It would be very beneficial to have new tools for the configuration, training, and user adaptation of the recognizer. The extensible feature recognition system allows further research into more sophisticated recognizers. It would be very useful to implement a machine learning algorithm within xstroke.

## 7 Availability

xstroke is free software distributed under the terms of the GNU General Public license (GPL). It is available from <http://www.xstroke.org>.

## 8 Acknowledgements

The most sincere thanks to my wife without whose patience this work would have never been completed.

Many thanks to Keith Packard for patiently explaining details of the X Window System, devising the technique for rendering the translucent, shadowed brush, and for suggesting the idea behind adaptive orientation correction.

The author owes a tremendous amount of sushi to Keith Packard and Bart Massey for tireless efforts at improving the presentation of this paper.

## 9 Disclaimer

Portions of this effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0529. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

## References

- [1] Jr. A. Chris Long, James A. Landay, Lawrence A. Rowe, and Joseph Michiels. Visual similarity of pen gestures. In *Proceedings of the CHI 2000 conference on Human factors in computing systems*, pages 360–367. ACM Press, 2000.
- [2] Jr. Allan Christian Long, James A. Landay, and Lawrence A. Rowe. PDA and gesture use in practice: Insights for designers of pen-based user interfaces. Technical report, U.C. Berkeley, UCB/CSD-97-976, 1997. URL <http://bmrc.berkeley.edu/research/publications/1997/142/142.html>.
- [3] David Bakhsh. Strokes package for xemacs, 1997. URL <http://www.mit.edu/people/cadet/strokes-help.html>.
- [4] Mike Bennett. wayV description (web document). URL <http://www.stressbunny.com/wayv/>.
- [5] Kalle Dalheimer. KDE: The highway ahead. In *Linux Journal*, number 58. Feb 1999.
- [6] Miguel de Icaza. The GNOME project. In *Linux Journal*, number 58. Feb 1999.
- [7] Yimei Ding, Fumitaka Kimura, and Yasuji Miyake. Slant estimation for handwritten words by directionally refined chain code. In *Proceedings of the Seventh International Workshop on Frontiers in Handwriting Recognition*, pages 53–61, Sep 2000.
- [8] Andy Edmonds and Pavol Vaskovic. Optimoz mouse gestures, 2001. URL <http://optimoz.mozdev.org/gestures/>.
- [9] Tyson R. Henry, Scott E. Hudson, and Gary L. Newell. Integrating gesture and snapping into a user interface toolkit. In *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*, pages 112–122. ACM Press, 1990.
- [10] Jason I. Hong and James A. Landay. Satin: a toolkit for informal ink-based applications. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 63–72. ACM Press, 2000.
- [11] James Kempf. Integrating handwriting recognition into unix. In *Proceedings of the USENIX Summer 1993 Technical Conference*, Jun 1993.
- [12] James A. Landay and Brad A. Myers. Extending an existing user interface toolkit to support gesture recognition. In *INTERACT '93 and CHI '93 conference companion on Human factors in computing systems*, pages 91–92. ACM Press, 1993.
- [13] Michael Moyle and Andy Cockburn. Gesture navigation: An alternative 'back' for the future. In *Extended Abstracts of ACM CHI'2002 Conference on Human Factors in Computing Systems*, pages 822–823. ACM Press, Apr 2002.
- [14] Keith Packard. Xkdrive manual. URL <http://www.xfree86.org/current/Xkdrive.1.html>.
- [15] The XFree86 Project. Xfree86 (web document). URL <http://www.xfree86.org>.
- [16] Mark Willey. Design and implementation of a stroke interface library. URL <http://www.etla.net/libstroke/>.



# Matchbox: Window Management Not for the Desktop

Matthew Allum  
*OpenedHand Ltd.*  
*London, England*

matthew@openedhand.com

## Abstract

Matchbox is a set of X11 utilities for managing applications. Matchbox is intended primarily for use on embedded X Window System devices with low display resolution, limited available input mechanisms, limited available storage and/or slow CPUs.

The core of Matchbox is an X window manager which aims to ameliorate shortcomings of existing desktop window managers on constrained platforms.

Matchbox approaches window management in a unique restricted way, benefiting the user, while striving to adhere to relevant standards such as the ICCCM and EWMH. Included applications include a PDA style application launcher, a panel and numerous panel applications. Matchbox also hopes to support emerging devices with limited input mechanisms such as Tablet PCs, HUD based devices and “wrist tops”.

## 1 Introduction

The past few years have heralded the wide availability of high-powered handheld computers such as the HP Ipaq. The processing power of such machines is equal to that of the desktop computers of just a few years ago. A typical device will have a 200MHz ARM processor, 32 Megabytes of RAM, and a 320 × 240 pixel touchscreen display.

During this same period, several factors have contributed to the emergence of open Unix-like software distributions for handhelds. These include the flexibility and platform independence of the software, the cooperation of hardware manufacturers, and the hard work of bedroom reverse engineers.

The standard windowing system for Unix-like machines is the X Window System [15], a powerful and flexible network-transparent window system. Client applications connect to the server, which performs the actual rendering of windows. A single client, known as the “window manager”, performs the task of managing other clients’ windows. The tasks of the window manager include framing windows with decorations, providing controls for common actions and managing window placement.

Unfortunately, many window managers do not cope well with a small display and limited input mechanisms

such as a touchscreen. Application window positioning and layout are often inappropriate, making the user interface awkward and unfriendly. In addition, specialized handheld computer input mechanisms, such as software keyboards, impose new requirements that often violate assumptions of conventional window managers.

With storage space at a premium, the selection of a window manager is greatly influenced by the size of its installed binary. Unfortunately, this constraint leads to use of simple window managers that lack features, visual flair and perhaps most importantly support for standards. Standards support is important so that the myriad of applications can expect a uniform interface for interacting with the window manager.

Matchbox includes a new window manager designed specifically for limited platforms. However, the Matchbox window manager also supports pre-existing and emerging standards. As much as possible, the Matchbox window manager attempts to manage existing applications in such a way as to make them more usable on small screens. It also provides features to enhance new applications that are specifically developed for handheld and embedded X platforms.

Matchbox also optionally offers features found in high-end desktop window managers including XML-based flexible theming, support for modern X infrastructure such as Xft [12] and the RandR extension [7] and support for utility libraries used by KDE [3] and GNOME [4] such as startup notification [13] and XSETTINGS [16].

Matchbox has grown to be more than just a window manager. It now includes a suite of tools for managing X11 applications. This paper will focus on the window manager, but will also discuss the included panel, desktop, utility library and panel applications.

## 2 X Window Management

Most window systems place responsibility for managing windows either within each application or within the window system itself. The X Window System [15] diverges from convention by supporting external window management: the geometry and stacking of windows on the screen is managed by a separate application known as the *window manager*. X Version 10 and earlier also

used external window management, but were unable to provide decorations and necessary event management to allow the implementation of friendly interfaces.

X Version 11 added several new mechanisms to support sophisticated external window management. Application requests for window placement can be *redirected* to the window manager: the window manager can then augment or amend the request as desired. Application windows can be *reparented*—placed within a window manager frame which also serves to contain window management user interface elements.

The basic X protocol avoids enforcing policy on applications. However, some level of cooperation is required for applications to successfully interoperate with a wide variety of window managers. The ICCCM sets the rules of engagement, so that applications know how to successfully interoperate with arbitrary window managers, session managers and peer applications.

The existing ICCCM standard may have been sufficient to describe operations in legacy X environments like CDE. New environments have included additional functionality not even conceived of in that era. A new organisation, freedesktop.org, was formed to set standards that extend the ICCCM. Standards promulgated by freedesktop.org add support for modern features like application docking, virtual desktop support, focus management, cooperative window management and additional window types. Many of these new features are very useful when managing limited screen space and input bandwidth. An important principle obeyed by all of these standards is that they do not specify precise behaviour. Rather, the standards describe required semantics of operations. As a consequence, applications and window managers are expected to accept any behaviour permitted by the specification. This is particularly important when developing novel window management techniques that dramatically change how windows are manipulated on-screen.

## 2.1 Basic Window Management

At the most primitive level, window management in X starts when an application creates a top-level window (known in X parlance as a “child of the root window”). The request from the application to make the window *mapped* (visible) is not acted on directly by the window system server. Instead, the window manager captures the window mapping request and performs whatever preparations may be necessary. These preparations usually include moving the application window to the interior of another window, then creating controls to manipulate the exterior window once it becomes visible.

Once the application window is prepared, the window manager makes it visible. The user can then manipulate the window through the controls drawn by the

window manager. Future requests to change the window’s size, stacking or visibility from the application are again redirected by the X server to the window manager. The window manager then performs the requested operations on the user’s behalf. X11 applications are expected to accept whatever position and size they receive from the window manager. For example, an early X window manager, the RTL Tiled Window Manager from CMU, would adjust the size and position of application windows to keep all windows fully visible on the screen.

## 2.2 Inter-client Communications Conventions

The basic window manager communication primitives of the X11 protocol suffice for simple applications. However, applications normally want the window manager to handle more sophisticated semantics. The core protocol provides no mechanism for applications to describe intended associations among windows and window modes. The ICCCM standard sets conventions for communicating this information.

ICCCM communication occurs through properties set on various windows, synthetic events generated by applications, and standard server-generated events. Clients set properties on their top-level windows to inform the window manager about various attributes. Common ICCCM properties include:

**WM\_NORMAL\_HINTS** Describes the set of desired sizes as a base size, a size increment and minimum and maximum sizes.

**WM\_HINTS** Describes the desired “state” of the window which is one of Normal, Iconic or Withdrawn, along with an icon to be associated with the window and any window group association.

**WM\_TRANSIENT\_FOR** Set for pop-ups to connect them with the associated main application window. Used for dialogs, not menus.

The CDE environment and Motif window manager (mwm) included additional properties on application windows to control mwm-specific decorations around application windows, such as the minimize/maximize control and window menu contents. These were not codified by the ICCCM, but have been widely implemented in existing window managers.

There are other properties holding window and icon name information but the basic ICCCM doesn’t provide any Unicode representation leaving these names effectively limited to ASCII or perhaps ISO Latin-1.

As far as a window manager is concerned, ICCCM compliance largely revolves around correctly interpreting application requests and sending the right messages back to applications. It has wide latitude in how to po-

sition windows on the screen and few hints on how windows are expected to be used.

## 2.3 Extended Window Management Hints

The KDE and Gnome project teams are working together on several standards to improve interoperability of applications and desktop environments. They adopted a set of ICCCM extensions proposed by Carsten Haitzler and Marko Macek and have published them as the Extended Window Manager Hints (EWMH) on the freedesktop.org web site.

The EWMH standard takes up where the ICCCM left off. EWMH sets policies and conventions to provide applications more control over how windows are managed on the screen. A key piece of additional information specified by EWMH is the classification of windows into broad semantic types:

**DESKTOP** A desktop window the size of the screen, placed beneath other windows. The window often contains icons that can be manipulated directly.

**DOCK** A dock or panel window. The window is often the width or height of the screen and placed along the edge to hold menus or other controls. The window is typically stacked above all other windows.

**TOOLBAR,MENU** A toolbar or pinnable menu window respectively (i.e. toolbars and menus “torn off” from the main application). The application may also set `WM_TRANSIENT_FOR` property on the window to mark the related application window.

**UTILITY** A persistent utility window like a palette or toolbox. These windows are different from toolbars: they are not “torn off” from the main application. These windows are also different from dialogs, because they are not transient and will probably stay open during work in the main application window. As with toolbars and menus, `WM_TRANSIENT_FOR` may be set.

**SPLASH** A window marking application startup.

**DIALOG** A transient dialog window. Windows without an EWMH type that have `WM_TRANSIENT_FOR` set are assumed to be of this type.

**NORMAL** A normal application window. Windows without either an EWMH type or `WM_TRANSIENT_FOR` are assumed to be of this type.

These types allow the application and window manager to cooperate in configuring the desktop environment and build different elements with separate applications.

EWMH also add many new window states beyond the Withdrawn, Iconic and Normal states specified by the ICCCM:

**MAXIMIZED\_VERT,MAXIMIZED\_HORZ** The win-

dow is maximized in the vertical or horizontal dimension.

**FULLSCREEN** The window should fill the screen without any visible decorations. A presentation program would use this hint.

**ABOVE,BELOW** The window should be stacked above or below most regular windows.

EWMH borrows an idea from the Motif window manager hints: applications can select which kinds of controls should be included in any window management decorations or menus. EWMH also allows applications to reserve space along the edge of the screen for panels and other controls. This instructs the window manager to avoid covering such windows when maximizing other windows. Finally, EWMH includes UTF-8 encoded window and icon titles to allow localized strings to be used. Window manager labels can thus be specified in a variety of languages.

## 3 Related Work

There are numerous window managers available for the X Window System, each with its own merits and unique features. Several window managers were investigated in terms of their suitability for use on a handheld device with limited application storage space and memory. The following window managers were considered: sawfish [9], blackbox [1], icewm [10], ion [17], and aewm [6]. The results of the investigation are described below.

Sawfish is a powerful, full-featured, programmable window manager. The power of sawfish comes at the expense of storage space requirements. While the core sawfish binary is only 128KB it has a large number of library dependencies, one of which contains a lisp interpreter. It is true that when compared with desktop software, sawfish would likely not be considered a large application. But its size becomes quite significant in a system with as little as 16MB available for application storage.

Blackbox is fast and visually appealing and depends only the core X libraries and libstdc++. Unfortunately it lacks support for modern EWMH standards. Its user interface relies on multiple mouse buttons—the right mouse button is used to access its root menu. It allows applications to initially size themselves larger than the actual display.

Icwm has good standards support. It does constrain application windows to the size of the display and is usable with a touchscreen, not relying on multiple mouse buttons. It also has a useful built-in panel. However navigating between windows on a small display is uncomfortable and its binary size is a rather large 480k.



Ion is small and has few dependencies. It is novel in that it tiles windows on the display. This makes good use of available display area as there are never overlapping windows. Unfortunately it is very dependent on keyboard control, does not always handle dialog windows well and has limited standards support.

Aewm is very small, has basic ICCCM compliance and only depends on core X libraries. However it is very basic both in functionality and visual appearance. Aewm relies heavily on multiple mouse buttons for its user interface making it nearly impossible to use with a touchscreen. It also allows for windows to resize themselves greater than that of the display. It is worth noting that Matchbox was initially based on aewm, though it now bears little, if any, resemblance.

Some of these window managers are of acceptable size and of nearly acceptable usability. As all are open source, they can be freely patched and adapted as needed. However none of them are ideal: the fact remains that no pre-existing window manager has been specifically designed for a PDA style device. The author's frustration with this situation has been the impetus for the creation of Matchbox.

## 4 Theory Of Operation

Consider the usage of a conventional desktop window manager on a constrained device with a  $320 \times 240$  display, no keyboard, and a touchscreen capable of only generating left mouse button events.

At best, applications too big for the display will be limited and resized to the full display size. However at worst, and more commonly, applications will get their requested window size and thus be obscured off screen. This is not a useful situation for the end user.

Assuming the application does fit on the display, or has been resized by the window manager to do so. It is very unlikely the window manager will provide any easily accessible mechanism to allow the user to then select between clients. Aids for this type of operation will be missing from the window title bar, a root window menu will be inaccessible and with no keyboard, key shortcuts are not possible. The user is left to awkwardly drag the top-level window off screen to reach lower level windows. With a large number of applications open this situation soon becomes unworkable.

There are other problems too, appropriate actions will not be taken to properly facilitate an input device window such as a software keyboard. For example, a software keyboard, to work well needs to be treated as a special case by the window manager and not the same as other application windows. It ideally needs not to overlap the application its being used to enter text into. If

this does happen it means unnecessary extra move resize actions for the the user. The keyboard also needs to never be given keyboard focus, otherwise it will end up sending key events to itself. While there is an ICCCM convention for specifying this, it is fairly obscure and not always implemented - for example blackbox does not.

Matchbox attempts to solve these problems primarily by managing window in a restrictive way. Restrictive management on such a device is good for the user. Actions, such as moving and resizing a window, are easy with a mouse but problematic with a stylus. If the display area is small these actions are needed repeatedly as the user struggles to move between overlapping windows. Therefore removing the need for the user to ever resize or move a main application window and handling it in a rudimentary way in the window manager provides an easier to use en system.

Matchbox organises applications in a manner similar to a deck of cards. Top-level application windows are requested and ultimately forced to take all available space, remaining locked to a static position. At any given time, only one such window is visible. The user is able to navigate through the deck of applications by various means. The window title-bar decoration may have buttons for navigating to the previous or next application in the window deck. There may be a button in the title-bar for a drop down menu listing an option for each application on the deck. Clicking an option will make that application visible. The presence, appearance and position of these buttons is configurable by the Matchbox theme.

Other navigation aids include keyboard shortcuts and external application task manager style programs. Any task manager that is EWMH compliant ( such as those included in GNOME and KDE ) will work.

This strategy is simple and restrictive in nature, but helps enormously on target devices trying to balance the applications needs with practical user control.

The strategy also fits in with the standards discussed above. While the application provides geometry hints to the window manager, the window manager may ultimately choose to set a different geometry.

An application window may be treated differently to the above if it provides standard hints or its properties match a certain criteria. Each of these mechanisms are discussed below.

### 4.1 Dialogs

Matchbox attempts to accommodate application dialog windows. Rather than being resized to fill the display like their parent application windows, their requested size is usually honored, (if within the display), and they are not statically positioned, allowing the user to drag them about the display.

Dialog windows are "paged" in the deck along with



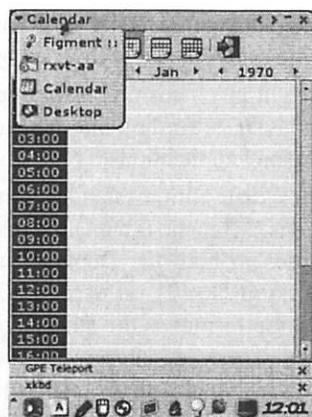


Figure 1: Ipaq screenshot showing Matchbox in action with various navigation aids.

their parent applications. Dialogs which do not have a parent, are permanently visible and are placed on the top of the window deck.

A dialog may be resized to fit the display, not cover panels or input devices if too large.

Matchbox uses the following criteria to decide if a window's properties nominate it to be treated as a dialog.

- The window sets its `WM_TRANSIENT_HINT` property.
- The window's `freedesktop.org` `_NET_WM_WINDOW_TYPE` property is set to `_NET_WM_WINDOW_TYPE_DIALOG`, or `_NET_WM_WINDOW_TYPE_SPLASH`.
- The window specifies that it is part of a group but not the group leader.
- Motif WM hints specify the window wants no decorations.

This last criteria is less reliable than previous ones. The rationale behind treating undecorated windows as dialogs is a heuristic based on the behaviour of many applications that use this kind of hinting. Many of these are shaped windows from applications such as `gkrellm` or `xmms` which simply break terribly when forced to resize.

## 4.2 Panels

A panel, or dock as it is sometimes referred to, is an area of the display used to hold small applications. These "applets" may be used to launch applications or provide information and notification to the user. Matchbox supports freedesktop compliant panels and places them along any edge of the display. They are always visible, and windows of other types are forced to fit around them.

A panel is identified when its `_NET_WM_WINDOW_TYPE` property is set to

`_NET_WM_WINDOW_TYPE_DOCK`.

## 4.3 Desktops

A desktop window is a full screen window with not title bar. It is positioned at the bottom of the window deck.

A panel is identified when its `_NET_WM_WINDOW_TYPE` property is set to `_NET_WM_WINDOW_TYPE_DESKTOP`.

## 4.4 Toolbar windows

Toolbar windows behave similar to the way user interface toolbars behave in applications such as web browsers. Toolbars are placed at the bottom of the display, below main application windows and above any panels. Their presence causes main application windows to resize accommodating them and they remain visible during application paging. Their width is set to the full display width and multiple tool bars are stacked vertically on top of one another.

Toolbar windows are collapsible with the window going into an iconic state and allowing the user to quickly free up screen real estate. This state is also easily reversible via buttons on the windows frame.

The purpose of tool bar windows is to provide accommodation for small utility type windows, specifically input devices such as software keyboards.

A tool bar is identified when its `_NET_WM_WINDOW_TYPE` property is set to `_NET_WM_WINDOW_TYPE_TOOLBAR`.



Figure 2: Ipaq screenshot showing a dialog and toolbar window.

## 5 Included Utilities

It is desirable for a window environment to include utilities for launching applications and managing instances of existing ones. The Matchbox window manager supports this to a degree via keyboard shortcuts and its built-in title bar task switcher.



Figure 3: Ipaq screenshot showing mbdesktop, mbdock and various panel applications

Existing applications that fill this role are either too big and unsuitable for embedded environments, or small but limited, lacking support for modern standards. For example the GNOME Panel provides a rich application management tools and much more. It is also flexible in usage allowing it, at least from a usability point of view, to be practical on a constrained device. However it has a huge range of large library dependencies instantly making it unsuitable when application storage space is at a premium. There are numerous simple toolkit based menu launchers available but many implement a unique non standard way of adding entries, are written in an unwanted toolkit on the target device, or lack support for X server extensions such as RandR. RandR is a modern X Server extension which allows for on the fly display resizes and rotations. This can confuse the positioning of older toolkit widgets that are not RandR aware.

To solve these problems, the Matchbox distribution contains a number of lightweight tools which are toolkit independent and adhere to freedesktop.org standards, allowing them to interoperate with environments other than just Matchbox. These include a panel and numerous panel applications such as system monitors and a menu based launcher, a PDA style “desktop” and a shared utility library.

## 5.1 The Panel and Panel applications - mbdock

The panel provides a small permanently visible area of the display to house small separate application windows. These applications are typically application managers, such as a launcher, system monitor or notification tools.

The docking mechanism works independently of the window manager and implements the SYSTEM\_TRAY[14] and XEMBED[5] specifications found at freedesktop.org. A panel application locates the panel by means of querying an X selection for a window ID

then communicating with a series of X messages and then finally being reparented and mapped by the panel.

This lightweight mechanism is also used to a degree in GNOME and KDE, affording interoperability of both the panels and panel applications

The Matchbox distribution includes numerous small panel applications. Mbmenu is a menu based application supporting entries in both the Debian “/usr/lib/menu” format and .desktop files[2], as used by GNOME and KDE. The .desktop format has the advantage of supporting internationalization and startup notification. There are monitor tools included with the distribution for memory and CPU usage, wireless signal strength, available battery power and a simple sound mixer tool. Numerous other third-party panel application are also available.

The panel also includes support for both multiple instances and multiple orientations.

## 5.2 The Desktop - mbdesktop

The included desktop is another application manager in a PDA style. Application icons and titles are placed in sectioned grid-like views. Mbdesktop is still in early stages of development with future plans being to allow extension to what is displayed by means of loadable modules. For example this could extend mbdesktop to do more than just application management - it could display browsable URL bookmarks, or specify an area of the desktop to provide PDA “Today” style summary textual information.

## 5.3 Shared utility library - libMB

LibMB contains useful shared code used by all included Matchbox utilities and optionally the Matchbox window manager itself.

Included code includes;

A small fast pixel buffer library for client side images. This library performs loading of PNG and XPM images, basic manipulation and composition. This part of the library is used by all Matchbox utilities and optionally the window manager.

The Abstraction of the XEMBED and SYSTEM\_TRAY protocols for easy creation of panel applications. It also safely extends the specification to allow alpha composition of panel applications on the panel.

A simple menu widget specifically designed for usage with touchscreens. This is used by both the panel and some panel applications.

A small .desktop file parser. Used by application launchers.

Various utility calls, for operations such as grabbing the root window image.

## 6 Flexibility / Tailoring to platform

Already there is quite a range in the capabilities of platforms supported by Matchbox. There are older and new cheaper devices which may be limited to a 4-bit greyscale display and 16MB of ram. There are also newer devices emerging such as the Sharp Zaurus SL-C700 which features a 640x480 16-bit display, a fast CPU, and ample memory.

It is desirable as a developer to take advantage of increased resources on newer devices, but less desirable for the end user to find his older device is no longer supported.

Matchbox attempts to keep all parties happy by the use of numerous compile-time options. These options produce numerous enhancements to operation but keep the core working essentially the same. The compile-time options allow for fine-grained control over library dependencies, features, binary size and memory usage.

A good example of this in action is the theming engine. People with higher-end devices will appreciate their window decorations being visually exciting and flexible—supporting multiple image formats, alpha blending and anti aliased fonts. Others however, may value their CPU cycles and storage space more precious and happily do without such features.

The Matchbox build system uses GNU autotools [18]. Using the `configure` script, the specifics of the created binaries can be controlled.

All build permutations provide the same core window management operations. Differences are primarily enriched visual look, and support for external libraries that provide features external to core window management.

With no options passed to the `configure` script, a conservative Matchbox will be built. Support for configurable theming is included though it lacks support for features like anti-aliased XFT text and PNG based images.

For a very tiny Matchbox the `'--enable-standalone'` `configure` option is used. This effectively replaces a large chunk of code implementing theming and XML parsing with a non-configurable theming system that uses only core X rendering functions to create a simple but fast look for window decorations. A standalone Matchbox also is dependent only on core Xlib libraries and no external configuration or image files.

A high-end Matchbox with most `configure` options enabled will give much richer theme support with support for PNG images, Xft anti-aliased fonts and freedesktop.org extensions such as XSETTINGS and startup notification.

## 7 Implementation

Matchbox is implemented in about 26400 lines of C code. Table 1 shows breakdown of the lines of code

Module	Lines of code
Window Manager	10744
libMB	5152
dock	4041
desktop	2872
7 applets	3574

Table 1: Lines of Code per Module

among the various modules of the window manager, the utilities, and the library shared between them (libMB).

Matchbox is written in a pseudo object-oriented style. The core of the window manager is an event loop which dispatches events to client objects as shown in Figure 4. Each client object represents an application window, and there is a separate client class for each of the supported window styles. Figure 5 shows the client class hierarchy.

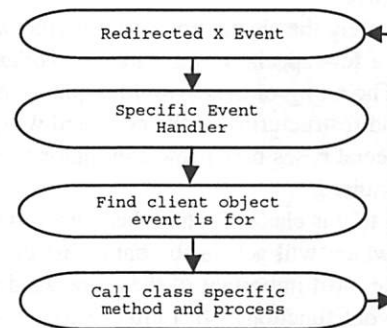


Figure 4: Matchbox event loop

All clients are stored in a circular doubly-linked list. The client structure is quite simple. In addition to basic client state information, (name, position, size, etc.), it contains pointers for the following class-specific functions:

- reparent
- redraw
- button\_press
- move\_resize
- configure
- get\_coverage
- hide
- show
- iconize
- destroy

The assumption is these particular methods should provide enough abstraction for the operations on various client types - though a client type may physically perform differently to the same method called on different client type. For example the iconize method will cause a main decked application to drop to the bottom of the deck, while if called on a toolbar application it will cause its toolbar to collapse.



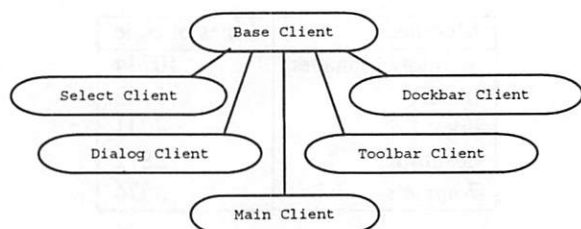


Figure 5: Matchbox client class hierarchy

A particular method or group of methods invocation can be mapped to the receiving of an event. For example on reception of an expose event, the client object for the event will be located and its redraw method called. Different window types need painting in different ways and structuring the code in this object-oriented way removes the need for lengthy if-then-else or switch-case code constructs.

Unfortunately the abstraction is not perfect and there does exist a few special cases where a workaround is required. The fixing of these would require a major reworking and restructuring of the code and with the discovered special cases being few and minor this has not yet been required.

Separate to this class structure there are various function calls which will act on the entire list of managed clients. The most important of these are window resizing and layout functions. If a client such as a toolbar resizes it will no doubt affect all other clients, so these functions manage keep the layout uniform.

There is a separate "theme engine" which performs the task of painting frame decorations. As discussed earlier, there are two interchangeable implementations of the theme engine, a flexible engine configured with XML configuration files and external graphics, and a standalone version designed for small fixed-use builds of Matchbox. Using the standalone engine reduces the size of Matchbox by about 25%.

## 8 Evaluation

Table 2 compares Matchbox with with other window managers previously mentioned.

Matchbox is able to achieve dramatic savings in size when compared to most window managers. Part of the savings is due to the restrictive window management style in Matchbox. Fewer supported styles of window management operations means less code bulk.

At the same time, Matchbox demonstrates that a window manager can be very small without going to the extreme minimalist style of a window manager such as aewm. Matchbox supports modern standards and provides high-level functionality with its theme support. The flexible build operations allow Matchbox to span a wide range of usage scenarios including static embedded

systems. This is a good model that can be followed by developers of many different classes of applications.

## 9 Future Work

The Author is currently happy with the core functionality and features provided by the Matchbox window manager. Improvements in the future will most like consist of minor improvements and bug fixes.

The project could benefit from some formal usability testing. Most user interface improvements and decisions have been personal decisions of the author or based on ad hoc feedback from users. Most user interface problems can be quickly solved with the flexibility of the theme configuration files requires no code changes.

Also investigation and possibly improvements to usage on newer larger constrained devices including Tablet PC's and set-top boxes. Initial experimentations on such devices have proved quite positive.

There is also the possibility or modifications for usage with future devices such as 'Head Up Display' based machines and so called 'wrist-tops'.

Most future improvements lie in the included utilities. Applications such as mbdesktop are still in early stages which much experimentation and newer features waiting to be done.

LibMB needs stabilizing with a solid API as well as documenting so it can be safely used by other developers.

There is scope for new included utilities, such as a small session manager or a panel based task switcher.

Also to avoid bloat, careful consideration has to be made before the addition of any new features that cannot be made as a compile time option.

## 10 Conclusion

Matchbox has proven very popular within the community of users of the Familiar[8] handheld environment. Familiar is a Linux distribution is for HP Ipaqs.

GPE[11], a GNOME like PDA environment based on GTK+ has chosen Matchbox as its core window manager. It too strives for standards support so the two fit together well. GPE developers have also written numerous panel applications.

I have also had reports of Matchbox being used with GNOME and KDE on desktop machines intended to be used by small children. Matchbox is also of value with these desktop environments on platforms such as Tablet PC's and personal video recorder style media boxes.

This success seems to suggest that design decisions made were correct.

## 11 Availability

Matchbox is free software released under the terms of the GNU general public license ( GPL ). It is



Name	Version	Size	Dependencies	Standards	Theme Support
sawfish	1.3	192K	X libraries, GTK+ libraries, librep and rep gtk bindings	ICCCM, EWMH	yes
blackbox	0.65.0	328K	X libraries, libstdc++	ICCCM	yes
ion	20020207	176K	X libraries	Limited ICCCM	no
icewm	1.2.6	480K	X libraries, Imlib	ICCCM	yes
aewm	1.2.2	24K	X libraries	ICCCM	no
matchbox	bells and whistles 0.5rc2	88K	X libraries, libpng, libxsettings, libstartup-notification, libexpat	ICCCM, EWMH	yes
matchbox	default build 0.5rc2	59K	X libraries, libpng	ICCCM, EWMH	yes
matchbox	standalone 0.5rc2	44K	X libraries	ICCCM, EWMH	no

Table 2: Window Manager Comparison

known to compile for Linux, various BSDs and Solaris. Releases, documentation and more are available at <http://handhelds.org/mallum/matchbox>.

## 12 Acknowledgements

My greatest thanks go to my fiancé, whose help, encouragement and patience have helped my ideas become a reality. Thanks also to Carl Worth, Keith Packard and Jim Gettys, whose advice and motivation throughout the project have been a great help. Finally, thanks to Keith, Carl, and Bart Massey for their help in producing the final draft of this paper.

## References

- [1] Adam A. Bellinson. The blackbox window manager. <http://blackboxwm.sourceforge.net>.
- [2] Preston Brown, Jonathan Blandford, and Owen Taylor. Desktop entry standard. <http://www.freedesktop.org/standards/desktop-entry-spec.html>.
- [3] Kalle Dalheimer. KDE: The highway ahead. In *Linux Journal*, number 58. Feb 1999.
- [4] Miguel de Icaza. The GNOME project. In *Linux Journal*, number 58. Feb 1999.
- [5] Mathias Ettrich and Owen Taylor. Xembed protocol specification. <http://www.freedesktop.org/standards/xembed.html>.
- [6] Decklin Foster. aewm. <http://www.red-bean.com/~decklin/aewm/>.
- [7] James Gettys and Keith Packard. The X Resize and Rotate Extension - RandR. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001. USENIX.
- [8] Alexander Guy and Russell Nelson. The familiar project. <http://familiar.handhelds.org>.
- [9] John Harper. sawfish: an extensible window manager. <http://sawmill.sourceforge.net>.
- [10] Marko Macek. icewm. <http://icewm.sourceforge.net>.
- [11] Colin Marquardt. Gpe: The gpe palmtop environment. <http://gpe.handhelds.org>.
- [12] Keith Packard. The Xft Font Library: Architecture and Users Guide. In *2001 XFree86 Technical Conference*, Oakland, CA, October 2001. USENIX.
- [13] Havoc Pennington. Startup notification protocol. <http://www.freedesktop.org/software/startup-notification/>.
- [14] Havoc Pennington. System tray protocol specification. <http://www.freedesktop.org/standards/systemtray.html>.
- [15] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.
- [16] Owen Taylor. Xsettings - cross toolkit configuration proposal. <http://www.freedesktop.org/standards/xsettings/xsettings.html>.
- [17] Tuomo Valkonen. ion. <http://modeemi.cs.tut.fi/~tuomov/ion/>.
- [18] Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf, Automake and Libtool*. New Riders, 2000. ISBN 1-57870-190-2.



# X Window System Network Performance

Keith Packard

*Cambridge Research Laboratory, HP Labs, HP*

keithp@keithp.com

James Gettys

*Cambridge Research Laboratory, HP Labs, HP*

Jim.Gettys@hp.com

## Abstract

Performance was an important issue in the development of X from the initial protocol design and continues to be important in modern application and extension development. That X is network transparent allows us to analyze the behavior of X from a perspective seldom possible in most systems. We passively monitor network packet flow to measure X application and server performance. The network simulation environment, the data capture tool and data analysis tools will be presented. Data from this analysis are used to show the performance impact of the Render extension, the limitations of the LBX extension and help identify specific application and toolkit performance problems. We believe this analysis technique can be usefully applied to other network protocols.

## 1 Introduction

The X Window System [SG92] was designed to offer good performance over campus area networks for machines and applications in common use in the late 1980's. A major part of the version 11 protocol design was to reduce the effect of network latency and bandwidth on application performance and correctness observed in analysis of X10 programs. Resource ID allocation was moved to the client applications to eliminate synchronous resource creation. Synchronous device grabs were added to provide correct operation of user interfaces when the user could manipulate input devices faster than the applications could respond. The selection mechanism was added to provide a uniform cut&paste model while delaying and optimizing the underlying bulk data transfer. Experience over the last 15 years has shown that further work in this area is possible and desirable.

There has previously never been packet level research capturing and analyzing the actual network characteristics of X applications. The best tools available have been dumps of proxy servers showing requests and responses, with no direct correlation with the actual packet level requests/responses and timing that determine network be-

havior (or on a local machine, context switches between the application and the X server).

One of the authors used the network visualization tool when analyzing the design of HTTP/1.1 [NGBS+97]. The methodology and tools used in that analysis involved passive packet level monitoring of traffic which allowed precise real-world measurements and comparisons. The work described in this paper combines this passive packet capture methodology with additional X protocol specific analysis and visualization. Our experience with this combination of the general technique with X specific additions was very positive and we believe provides a powerful tool that could be used in the analysis of other widely used protocols.

With measurement tools in hand, we set about characterizing the performance of a significant selection of X applications based on a range of toolkits and using a variety of different implementation techniques. We are specifically interested in the effects of the Render extension's [Pac01] approach to text, and whether the optimizations offered by the Low Bandwidth X (LBX) extension [FK93] [Pac94] were of any use. Specific application performance issues were also discovered and forwarded to the relevant open source projects.

## 2 Measuring X Performance

The measurements taken span a range of applications, network characteristics and protocol compression techniques.

### 2.1 Selecting Test Applications

An attempt was made to select applications representative of modern X usage. Reasonably current versions of each application were selected from the Debian Linux distribution. One advantage of the passive monitoring technique is the ability to monitor several applications simultaneously and measure the performance of the collection as they interact on the network. This permits the analysis of session startup sequences where many applications are competing for resources.

The following applications were measured in this ini-

tial work

- Mozilla (version 1.3); Web browser with client-side fonts, imaging the microscale synthetic web page developed for the HTTP/1.1 performance work
- Mozilla (version 1.3); Web browser with server-side fonts, imaging the microscale synthetic web page developed for the HTTP/1.1 performance work
- Kedit (Version 1.3) simple text editor based on the Qt toolkit
- Nautilus (Version 2.2.2) file browser based on the GTK+ toolkit
- KDE Session (Version 3.1) full KDE session startup.

### 2.1.1 Mozilla and Fonts

Because of the rapid uptake of client-side fonts in application development, Mozilla was the only current application available that supports both core and client-side fonts in the same version. The current Debian Mozilla package provides an alternate version of the key drawing library that controls access to fonts; the only change in the Mozilla configuration required to select which style of fonts to use is to replace this library with the appropriate version. This limits the changes as much as possible which should make the resulting measurements an accurate representation of the difference between the two techniques.

### 2.1.2 KDE Session and LBX

The XFree86 4.3 releases of the LBX extension and proxy are faulty, causing the KDE session to hang about halfway through the startup process. Some modest attempts to discover the problem did not yield any results and so those values don't appear in the resulting graphs.

## 2.2 Network Performance Characteristics

For this study, one goal was to measure the performance impact of latency vs. bandwidth, and so a set of latency and bandwidth values were used and each test run with every pair of latency and bandwidth. The five bandwidths used were 100Mb, 10Mb, 1Mb, 100Kb and 10Kb. The four latencies used were 0.1ms, 1ms, 10ms and 100ms. These latencies represent the time for a packet to traverse through the router in one direction, and so a round trip time would be at least twice that value.

## 2.3 X Protocol Encoding

The X protocol was originally designed to run efficiently over the campus area networks available in the late 1980's, which ran at 10Mb/second. As modem performance improved in the early 1990's it became almost feasible to run X applications over dial-up links.

LBX was an effort to improve performance over low-speed/high-latency links. A significant amount of that work was focused on reducing latency effects of slow links without modifying applications. Another major part of the work was in creating custom encodings of X data to reduce the bandwidth required. The latency efforts focused on problems seen when LBX was being designed. How well those efforts carry forward to modern applications is explored in this paper.

More recently, network proxy support has been added to the SSH [BS01] protocol enabling X connections to be forwarded through a secure and optionally compressed connection. SSH has no X-specific re-encoding or compression techniques, it simply uses the Gzip [Gai93] compression technique on the datastream when compression is enabled.

Measurements using raw X protocol as well as X run through both of these proxies show their effectiveness in improving performance.

## 3 Passive Network Analysis

In the past, ethernet data monitoring required very little equipment; any host could trivially monitor traffic on the wire from any other host by placing the ethernet hardware in promiscuous mode. Such is not the case today; twisted pair ethernet performance depends on switched connections to provide a collision free link between the two endpoints. The easiest way to monitor the network is to capture packets on one end of the connection. While this may consume some CPU resources, the overhead is minimal given modern machine performance. Alternatively, a "man in the middle" machine can be used to capture packet traces between client and server. There are also network switches which can mirror traffic from one port to another, but a single connection cannot reliably hold traffic from a full-duplex link and so packets might be lost or packet ordering scrambled. For either host-based technique, systems are now fast enough that it is possible to capture all packets in most applications without drops. We chose to capture packets on the X client end of the network approach as the network simulator caused problems with capture at the router. The passive capture configuration can be seen in Figure 1.

### 3.1 Network Data Capture

To minimize the impact of disk activity on the network monitoring process, the packet capture tool provides minimal information about each network event that is seen in the protocol trace. Each trace record is timestamped with that captured by the kernel packet logging facility which provides kernel-level timestamping accurate to well under a millisecond. The capture application is run at nice -20 which has proven sufficient to avoid packet loss.



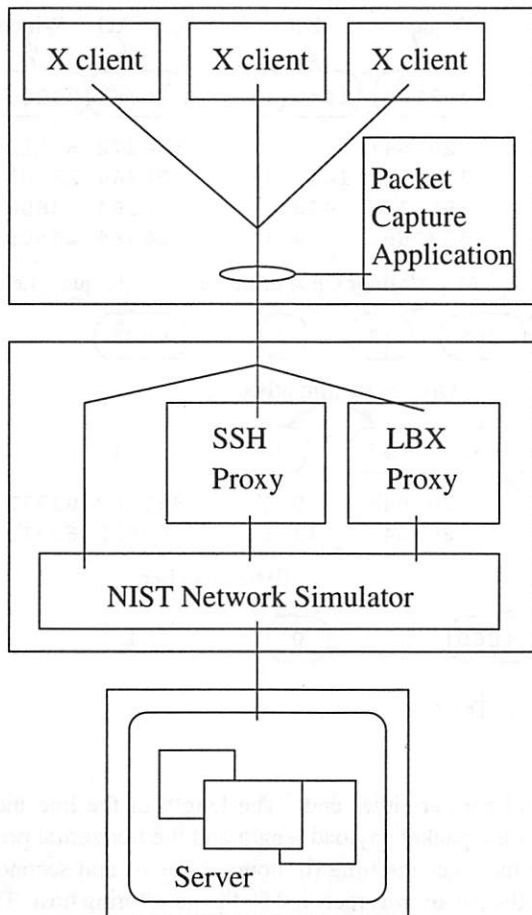


Figure 1: Network Capture Configuration

A low level trace of the raw IP packets is logged and includes the sequence numbers, window size and packet length. This data is used to deduce the raw network state and determine whether the network is busy or whether one or both ends of the connection are compute bound.

Above that, a trace of the exchanged X messages is logged. Each message is timestamped with the packet time containing the end of the message along with the X sequence number and length. Requests include the request id, events and errors include the appropriate code. If no data packets are lost in the monitoring stream, the X protocol can be completely reconstructed by the capture application. A sample of the raw data file can be seen in Figure 2.

The data are logged in this minimal form to disk and the analysis is performed off-line. This raw logging serves both to reduce computational load during capture as well as to preserve the traces for multiple analysis mechanisms and regression testing of the analysis programs themselves.

### 3.2 Simulating Network Conditions

The NISTNet package [Gro00] converts a Linux machine into a network emulator capable of simulating a wide variety of network conditions. NIST Net is designed to emulate end-to-end performance characteristics imposed by various wide area network topologies, such as restricted bandwidth, increased latency or even packet loss.

While this package provides the ability to model inconsistent networks, we chose to simplify the data analysis in this work by modeling networks with constant bandwidth and latency.

For the 100Mb test, NISTNet was disabled and the middle machine configured to act as a simple router.

## 4 Network Performance Analysis

With data captured to disk files, a pair of tools were used to visualize the data and compute quantitative performance results from them.

### 4.1 Application and Network States

To examine the application and network behavior with a greater level of detail, a simple model of the operation of the application and the X server was developed. In each direction, the network can be in one of a few states:

- Idle. The network can accept additional data without delaying the transmitter.
- Bandwidth Limited. The rate of data transmission is essentially equal to the available bandwidth.
- Window Limited. The receiver window is full and the transmitter awaits a response advancing the window.

Idle periods are separated from congested periods by noticing when acknowledgement packets immediately elicit additional data from the transmitter. The assumption is that the transmitter was blocked since the delivery of the preceding data packet. When the transmit window is closed, the assumption is that the receiver is blocked for some reason, rather than the network. When running X through a proxy, the receiver is the proxy application which is blocked when the low speed link is congested, so the distinction between the two congested states on the low speed link cannot be determined by examining packets captured from the high speed link.

In addition, the time applications spend waiting for the X server to produce a reply is measured by checking requests which have replies and which are delivered just before a reply is received.

The sum of the time spent waiting for transmission and replies is referred to as the Total Network Delay. This is a slight overestimate as time spent waiting for replies can also include a portion of the time spent waiting for transmission space.

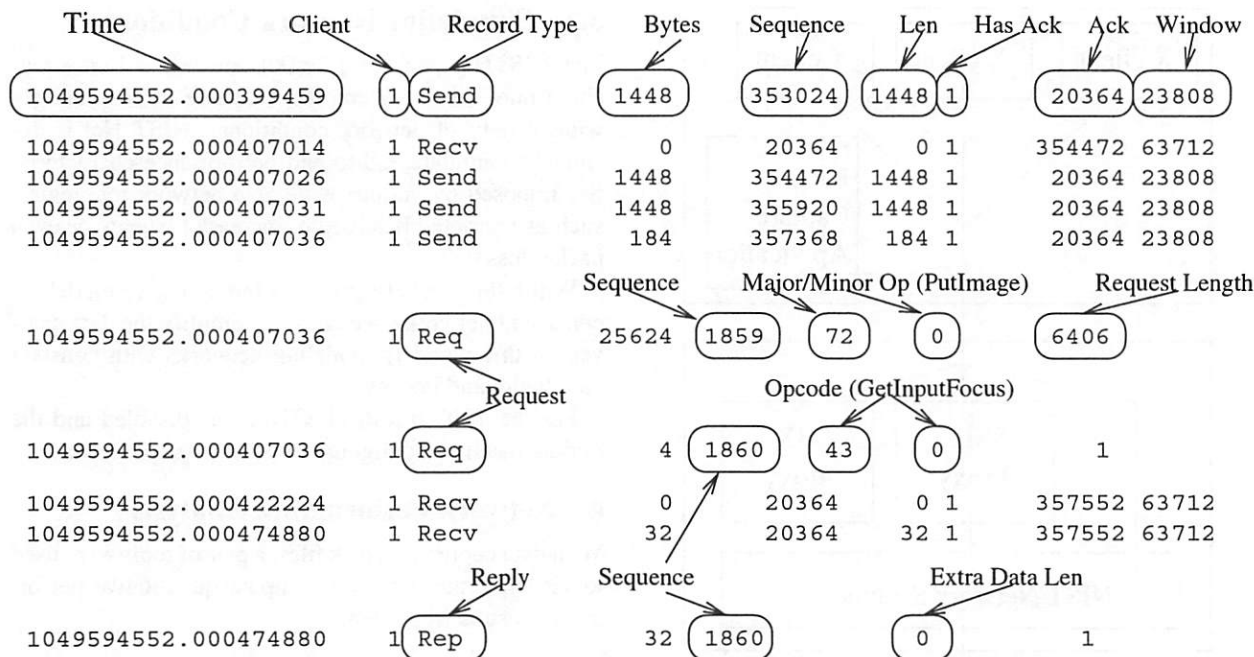


Figure 2: Sample Packet Trace

## 4.2 Network Visualization

A raw packet trace is difficult to analyze manually; it consists of simply a stream of numbers, and the most important data is contained only in the differences between them. In his 1990 master thesis, Tim Shepard describes the xplot network performance visualization tool [She90].

We renamed this tool 'netplot' in our environment because of conflicts with another application named 'xplot', it is otherwise unchanged from the xplot source code.

Even the immediate glance at the resulting netplot is useful, as seen in Figure 3. If the X server is always faster than the client (the case over any network studied here), an ideal application would never be latency bound and would have a constant slope. No real application, of course, is ideal. Any approximately horizontal region of a plot is either the application unable to provide data because it is busy for some other, non window system related reason, or, the application is waiting on requests from the X server. The user can then quickly zoom in on these areas to determine the reasons for "less than ideal" behavior. The slope of regions of the graph which are network limited can be used to determine the actual effective bandwidth.

More detail from the netplot tool can be seen by inspecting closer detail of the time and sequence number plot of a network as seen in Figure 4. As in the original xplot paper, each packet is drawn as a line with hori-

zontal bars at either end. The length of the line indicates the packet payload length and the horizontal position indicates the time (in hours, minutes and seconds) that the packet was received by the monitoring host. The acknowledged data and available window are drawn by connected sequences of horizontal and vertical segments surrounding the packets themselves.

In addition to the original presentation of the raw network performance, each packet is broken down into X messages drawn with lines and arrow heads and marked with the name of the X request, error or event. These are shown at the time of the packet containing the end of the message.

When operated interactively, the state of the connection is marked in the color of the lines and arrowheads. These markings serve both to highlight areas with possible performance issues as well as to verify that the categorization algorithm is operating as expected.

## 4.3 Quantitative Network Performance Analysis

A separate tool, xcapanalyse, produces quantitative data about the performance of the connection which is summarized by a small table including one row for the request stream and another for the response. An example from the kedit test can be seen in Table 1. In this table, there are three separate X clients running simultaneously; kedit launches two KDE helper applications when run on an otherwise idle X server. The network

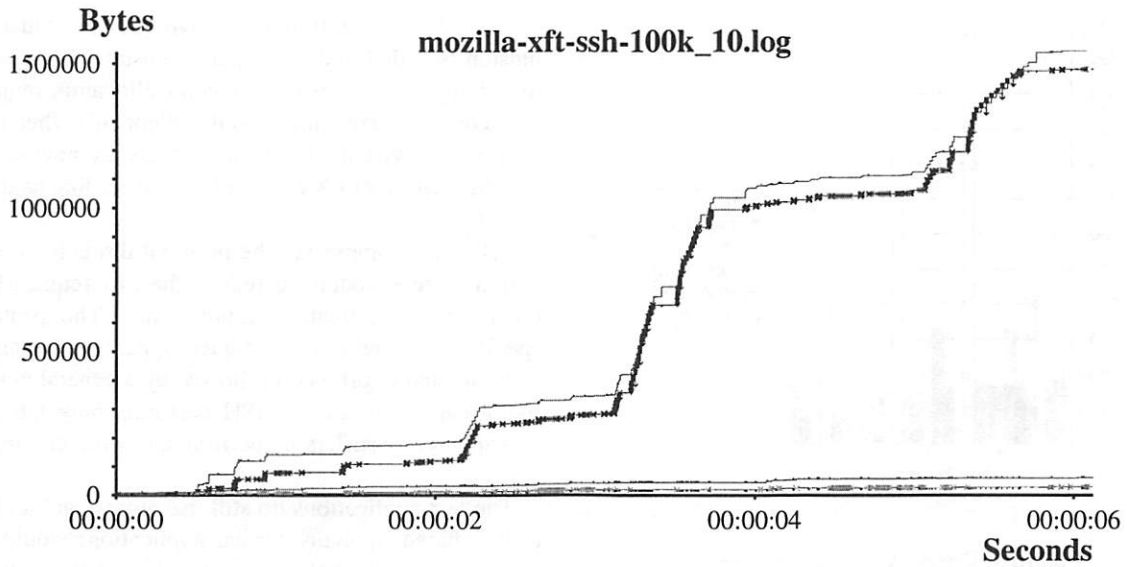


Figure 3: Mozilla Trace

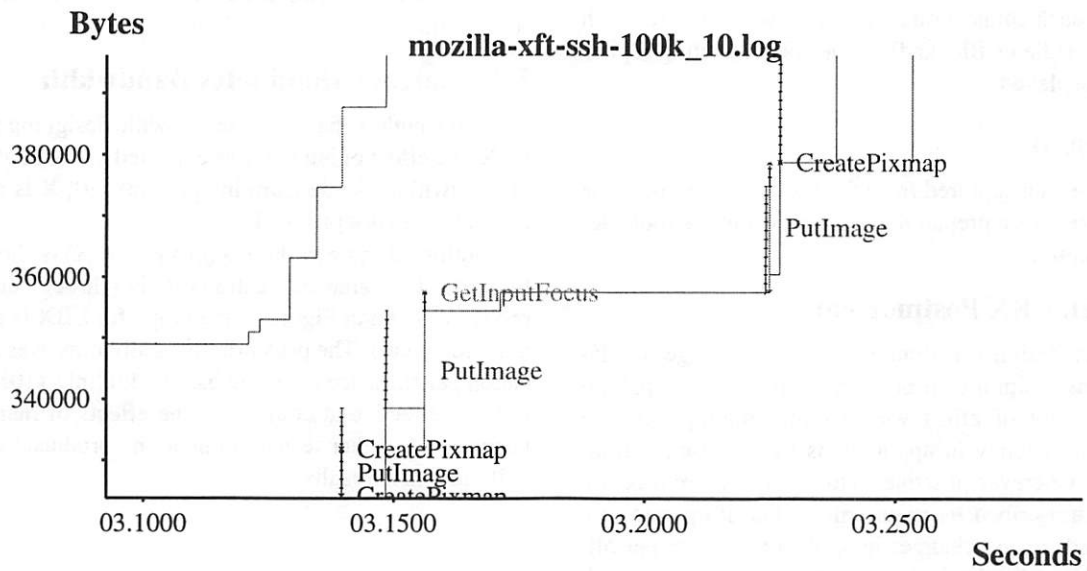


Figure 4: Mozilla Startup Trace

ID	Direction	Idle	Block Send	Block Reply
1:	request	2.901014	0.012351	0.883300
	reply	2.844048	0.032043	
	total	2.868971	0.044394	0.883300
2:	request	0.660025	0.000613	0.338392
	reply	0.628917	0.000123	
	total	0.659902	0.000736	0.338392
3:	request	0.091822	0.000844	0.043864
	reply	0.055788	0.000000	
	total	0.091822	0.000844	0.043864
All:	request	2.901802	0.011563	
	reply	2.843925	0.032166	

Table 1: Network State Analysis for Kedit

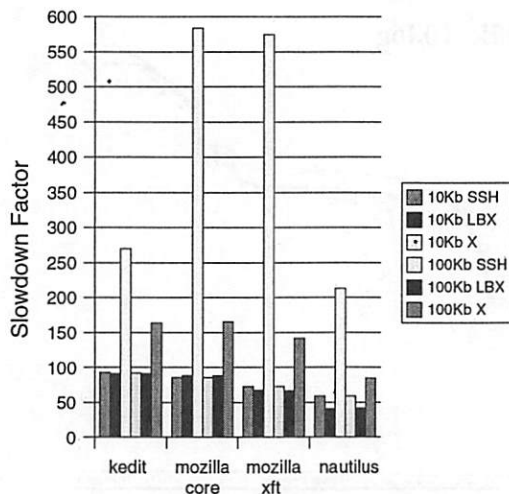


Figure 5: Proxy Performance Effects

state for each connection is split into two values for each direction (Idle or Blocked). Time spent awaiting replies is also displayed.

## 5 Results

Using the data captured from the test applications, some hypotheses were prepared and tested using the tools described above.

### 5.1 An LBX Postmortem

The LBX design was done with the knowledge that latency was a significant problem in running X applications. A lot of effort was put into finding ways to ameliorate latency in applications by 'short-circuiting' requests wherever possible. However, the architecture was circumscribed by requirements that it operate as a proxy and that no changes in applications were permitted. LBX only eliminates round trips for replies with unchanging data, such as QueryExtension, InternAtom and GetAtomName.

Figure 5 shows the effect of LBX and SSH on a variety of applications over a network with 100ms latency. The values displayed represent performance relative to a 10Mb ethernet link in terms of total network delay; kedit experiences slightly more than 250 times as much network delay over a 10Kb/100ms link than it does over a 10Mb/0.1ms link. This chart demonstrates that LBX does not perform any better in these environments than SSH. The 100ms latency tests yielded the best LBX results of all. At lower latency values, LBX performed worse than SSH for all applications.

LBX latency mitigation techniques are limited to color allocation, atom management and the QueryExtension request. Of these, only the atom requests see

any significant use in modern environments. QueryExtension is called only once per extension. None of the tested applications made any color allocation requests, they compute pixel values on the client side when using a TrueColor visual. TrueColor visuals are now used in the vast majority of X environments, including handheld devices.

LBX also compresses the protocol through a combination of re-encoding to reduce the raw request byte-count and bytestream compression. The protocol-specific compression techniques appear to be an entirely wasted effort when followed by a general purpose bytestream compressor; SSH performs only the latter and apparently suffers no performance problem as a result.

The test applications do still use atoms, and an LBX proxy shared by many similar applications would usefully cache these values on one side of the network. However, Qt applications demonstrate how to fix this by pre-interning atoms used within each application at startup time.

### 5.2 Latency Dominates Bandwidth

Given the authors past experience while designing X and LBX, the effect of latency was expected to dominate that of bandwidth. Aside from image transport, X is a very compact network protocol.

Nautilus running without a proxy (raw X) is shown in Figure 6. The same application running through an SSH proxy is shown in Figure 7 (the graph for LBX is essentially identical). The proxy dramatically improves application performance over low bandwidth links (10Kbps), but is ineffective at countering the effects of increased latency. The other tested applications produced essentially identical results.

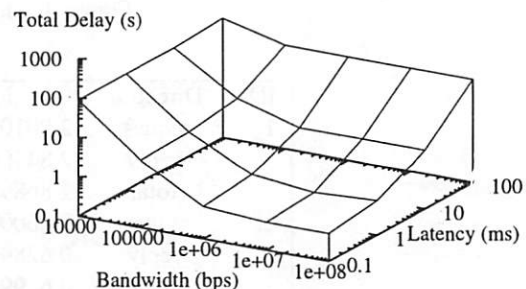


Figure 6: X Total Delay (Nautilus)

Network bandwidth and latency both affect X application performance. The question is how they relate.



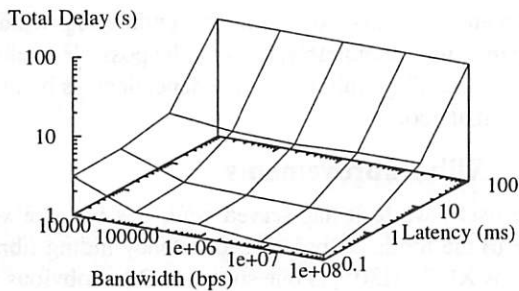


Figure 7: SSH Total Delay (Nautilus)

Request	Count
InternAtom	46
GetInputFocus	20
GetProperty	15
GetGeometry	13
QueryExtension	10
XKB_UseExtension	4
GetSelectionOwner	4
TranslateCoords	3
QueryTree	3
XKB_PerClientFlags	2
XI_OpenDevice	2
XI_ListInputDevices	2
RenderQueryPictFormats	2
ListExtensions	2
GetKeyboardControl	2
BigReqEnable	2
Total	132

Table 2: Synchronous Requests in Nautilus

For HTTP or FTP transfers, network latency has little or no effect on performance as TCP is designed to mitigate against latency by increasing window sizes as needed. Interactive protocols like SNMP, POP3 or X cannot mask latency in the same way as responses are computed from requests.

X11 was designed to reduce the effect of latency as seen by X10 applications in a few ways – resource allocation was moved from the X server to the X client which resulted in a huge improvement in application startup performance. However, Xlib still exposes many synchronous APIs which pause application execution to wait for a reply. Looking at the raw request traffic from Nautilus shows that while starting up, it waits for 132 replies as seen in Table 2. Minor changes in the application could eliminate a significant number of these.

Compressing the X protocol with a general purpose algorithm solves the network effects due to bandwidth down to a 10Kb link. Application changes to reduce the number of synchronous requests can eliminate much of the dependency on latency.

### 5.3 Using Atoms

Gtk+ applications spend considerable time getting atom values from the server, as shown above in Table 2. Nautilus makes 46 separate synchronous calls for atom values. Qt demonstrates that by pre-caching expected atoms, the number of synchronous requests can be dramatically reduced. Kedit makes 104 requests for atom values and yet waits only 6 times.

### 5.4 Client-side Font Performance

The most radical recent shift in the X Window System has been the migration of font support from the X server to the application. The Render extension provides only glyph storage and rendering functionality; all font access and glyph rasterization is done by the client. While there are many good architectural reasons for this shift [Get02], it represents a major change in how text operations appear on the network.

Two otherwise almost identical versions of Mozilla were installed and configured, one using the core server-side font APIs and the other using client-side fonts. Figure 8 shows total network delays for a range of network latencies at a fixed 1Mb bandwidth for core fonts and client fonts. The client-side version is consistently faster than the core version.

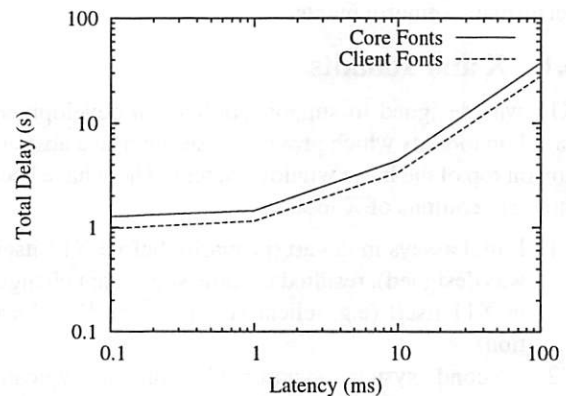


Figure 8: Client vs Server Fonts (Mozilla)

The core-font version fetched 64KB of font data from the server while the client-font version sent 62KB of glyph data to the X server. Early estimates during Render design discussions about the glyph images sent by client side fonts balancing the glyph metrics received from server side fonts are borne out by this example.

More significantly for low performance networks, the server-side font version required 40 additional round trips to list fonts and retrieve font metrics from the server which resulted in the savings measured above. Client side fonts have now removed one of the largest latency offenders, reducing total network delays by roughly 25% during application startup.

## 6 Improving X Performance

While the main subject of this research was to produce a methodology for measuring X network performance and answering general questions about what application and protocol features affect performance, the data collected also highlight some performance issues for each of the applications run. The results in the previous section point to where efforts should be focused in attempts to improve X performance.

These possible changes include:

- changes in toolkits
- extensions to the X library to hide latency
- extensions to X itself, including bandwidth conservation

Reducing network delays due to latency can only be effectively done by changing applications. Many of the changes needed can be done inside the toolkits which will provide benefit to many applications. Other changes would benefit from changes in underlying libraries to make latency hiding easier. Finally, while latency is the largest issue in performance over slow networks, it is interesting to consider whether bandwidth reductions through compression could be usefully applied in higher performance environments.

### 6.1 X and Toolkits

X11 was designed to support application development based on toolkits which provide a user interface abstraction on top of the basic window system. There have been three generations of X toolkits:

1. Initial assays in the art (primarily before X11 itself was designed), resulted in some significant changes in X11 itself (e.g. client side resource ID allocation).
2. "Second system syndrome" toolkits, typically based on Xt [AS90] such as Motif have had some performance work done on them using xscope and similar tools, though the lack of detailed time information failed to highlight the importance of latency elimination.
3. New toolkits, such as Qt [Dal01], GTK+ [Har99], and Mozilla [BKO<sup>+</sup>02] which operate at a higher level of abstraction than 2, as they add sophisticated text layout, canvases and image abstractions to applications, while hiding detailed information about

the display's characteristics from clients. These toolkits are only now seeing serious scrutiny.

Because toolkits now hide the underlying window system almost completely, it should be possible to eliminate almost all gratuitous latency dependencies by modifying that code.

### 6.2 Xlib Improvements

Xlib itself, while it has served well, has become very long in the tooth. A redesigned, latency hiding library such as XCB [MS01] is one solution. Other obvious additions to Xlib can and should be implemented in short order, for example, a call back based XGetProperties interface would enable toolkits to hide much of the latency when communicating with other clients. The Metacity window manager found this technique valuable, and it is planned for the GTK+ toolkit, to mitigate problems in drag and drop as well as startup time. As this is very common and involves Xlib internal interfaces, this should be added to Xlib for general use.

One of the design "mistakes" of X11 exacerbated by its very success is the extensible type system called atoms (as in the Lisp systems from which it was derived). This has been heavily used in the interclient communications protocols used between applications (primarily toolkits) and window managers. The InternAtom function requires a round trip to provide agreement among clients on a small (32 bit) handle for a string. A modern design would almost certainly avoid round trips entirely by using cryptographic hashes (or just using strings everywhere). Unfortunately, it is very hard to retrofit this, as atom values are so small that collisions on common hash functions would be common enough to be worrisome. Toolkits and applications, however, generally know the names of all atoms they use and this can be (sometimes with some pain rewriting code) reduced to a single round trip hiding most or all of the latency, transport, and context switch overhead. Alternatively, we could have an extension to return all atoms and names in one request. Further investigation is in order.

X11's extension system is a minimalistic design, intended to mitigate the major problems that occurred as X10 was extended. Extension control is inadequate, having no facility to request reload of extensions and lacks any generic version mechanism. There are now about 10 extensions used by toolkits: querying for them and instantiating them has become a significant part of the startup time of applications, and becomes more so as other the latencies are reduced. One round trip is required to get the protocol op-code, and typically a second round trip to get a version number of the extension, resulting in approximately 20 round trips. Ironically, implementing an extensions extension may well prove the

best solution, both recovering the latency and providing the missing extension functionality, rather than just batching multiple requests, the other obvious solution.

### 6.3 Bulk Data Compression

Bandwidth is another issue worth further study. While latency may dominate application performance today, that can be solved without changes to the underlying protocol. It would be useful to investigate whether new encoding or compression techniques should be included in the protocol in the near future as such changes involve both sides of the wire and take considerably longer to deploy than changes on only one side or the other.

The compression available from SSH already demonstrates some benefits for low bandwidth links. However, SSH increases latency on high speed links as it adds context switches and several manipulations of the data. Image and glyph data are often the only significant bandwidth consumers in X applications, and so they provide an easy target for specific compression operations directly within the protocol libraries.

Figure 9 shows a KDE sending the desktop background through an SSH proxy. SSH bytestream compression allows the 10Kb link to transmit 3.4MB of raw X request data in 7.99 seconds, an effective bandwidth of over 3Mbit/second (!). X images are usually extremely compressible in their raw form.

But bandwidth is not only a concern when using X over a network. Modern graphics chips are usually bandwidth bound: they use all available bandwidth of the AGP bus for transmission. And the general architectural trend of the last 15 years has been that CPU cycles have increased faster than bandwidth. Additionally, keeping everything in cache has become increasingly important for best CPU performance. Keeping the protocol compact also minimizes context switches involved in data transport between applications and the X server. All these trends argue toward much closer conservation of bandwidth.

There are (at least) four approaches, some of which are particularly useful in the local case, and some in the network case.

- careful design of protocol extensions
- additions to the X protocol to introduce protocol specific compression tricks
- datatype dependent compression
- the use of general purpose stream compressors such as ssh.

In the core protocol, GC's (graphics contexts) were used to good advantage to reduce the size of graphics requests. More recently, the Render extension was designed to avoid redundant transmission of both window and graphics state, as, arguably, the X core protocol

should have been designed originally. Not transmitting redundant information is certainly better than having to compress that information.

The Render extension replaces high-level geometric objects with numerous low level filled polygons (either triangles or trapezoids). An open question at this point is whether the large number of these primitive objects will have a measurable performance impact for X applications. Most X applications draw very few geometric objects (lines, circles, polygons) of any kind, but particular applications may perform very intensive graphics operations.

The Render trapezoid encoding is quite general and it is likely that when tessellating shapes to trapezoids, many of the coordinates will be repeated among multiple trapezoids. A simple addition to allow requests to reuse recently transmitted information in following trapezoids may reduce the amount of data transmitted more efficiently than a general purpose compression system could. This would reduce overall memory traffic and will even be a significant benefit in the local case when trapezoid rendering is accelerated in hardware.

Currently, the Render extension transmits glyphs in an uncompressed form to the server. With most applications and at the current typical screen resolution, the amount of data transmitted is roughly comparable with the use of core fonts as font metrics no longer need be transmitted. The glyphs, however, are highly compressible. Additionally, as screen resolution scales up, the size of glyphs will increase, though if compressed, the bandwidth will scale sub-linearly with resolution. Consequently, glyph compression may well be included in a future Render extension version.

Similarly, a compressed image transport extension would make a significant difference for many applications (such as web browsers), as most of the data is images, and often already available in compressed form (e.g. GIF, PNG) within the client.

We will need to explore the tradeoffs between CPU usage and compression efficiency further before deciding what algorithms are most appropriate, while avoiding the image processing, over-engineering and design problems that made XIE useless in practice and one of the most notable failures in attempted X extensions.

Gian Filippo Pinzari [Pin03] has recently pointed out that Xlib does not zero out unused bytes in the protocol stream but transmits the left over bytes in the protocol buffer where the requests are formed. Doing so properly would not increase memory traffic given write-combining cache architectures, and may help the efficiency of general purpose compressors such as SSH. Mr. Pinzari reports significant improvements in compression ratios.



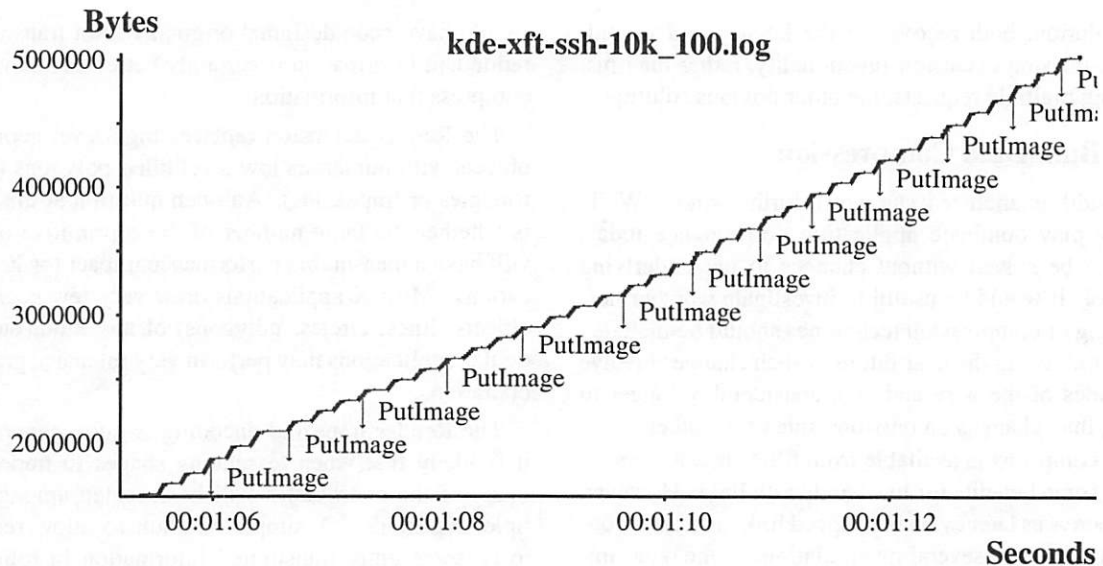


Figure 9: Effect of Compression on Bandwidth

## 7 Conclusions

Performing protocol-level performance analysis combined with passive network monitoring provides a new way to capture the effects of the network on application performance. This new data demonstrates some of the effects of recent changes in the X Window System environment and provides critical direction in improving application performance over network connections.

This tool and methodology has allowed for quick understanding and quantification of performance issues.

- Client side fonts are making a significant reduction in round trips, as expected, with real applications. Application startup time is significantly reduced, relative to using core fonts, in all cases, (local machine, local net, or simulated broadband network). This has demonstrated consistent 25% reductions in total application startup delays due to the network.
- Bandwidth usage for client side fonts is about the same as with server side fonts, as expected, even with the glyphs uncompressed. Compressing glyphs for transport would help greatly.
- Performance using LBX over the highest latency links is no better than SSH. Performance over broadband network of the protocol without compression is only 25 percent worse than LBX. LBX does not solve either the authentication and security problems that SSH solves. We saw little evidence of LBX ever helping. At least as implemented, LBX looks to have been a bad idea.
- Some of the new toolkits are performing unnecessary round trips: we are working with the appropriate projects to get them fixed.

- InternAtom is a hotspot in GTK+ based applications. Qt has already solved this problem by pre-fetching many likely atoms at application startup.
- GetProperty and other inter-client communications mechanisms is a minor problem and might see some relief through asynchronous Xlib interfaces. The Metacity window manager has already demonstrated this technique with favorable results.
- Typically, with current applications based on current toolkits, 10 extensions are initialized, and these typically require at least two round trips before they can be used. A solution akin to the GetProperties suggestion above could work, but we feel an extension extension would be more productive since there are other shortcomings with X11's extension system.
- Image transport remains the dominant consumer of network bandwidth. Client side fonts increase the number of images transmitted by including glyphs in that mix.

For glyphs, further experimentation is needed to understand exactly what compression algorithm would be best under what circumstances, and the relative tradeoffs of compression factor versus CPU time overhead.

Image transport itself is a very common operation, and could enable significant memory, CPU savings, and speed networks.

This is in part due to image sizes having grown due to much greater depth. Most users have 32 bits/pixel in 2003, rather than the one or eight bits/pixel during X11's design, and usage has become much more image dominated, compounding



the issue.

The most common applications have the images already in compressed format. If X can accept the images in original formats, then memory can be saved in many clients, wire transmission time reduced, CPU time saved in transport, and applications sped up when used remotely from the display.

Focusing entirely on image transport rather than display may allow design of such an extension to avoid the traps that made the XIE extension worthless.

For some interesting applications (e.g. Mozilla), a factor of 10 reduction in startup delays due to the network may be possible on broadband links through a combination of compression and latency amelioration techniques.

Many of these improvements would improve performance in the local case as well. Elimination of unneeded round trips reduces context switches. Avoiding gratuitous decompression/compression stages reduces memory space and bandwidth consumption. Sharing common vertex data could reduce the amount of data sent to the graphics card. Measuring the effect of such changes in a systematic fashion will require data collection techniques than presented here, but the same kind of data analysis could be used.

## 8 Future Performance Analysis Work

The methodology and tools described in this paper are a great aid in understanding the performance of X applications, and we believe could be very useful for other application protocols.

One could build a set of tools generic to applications protocols, as the methods used would work for most protocols with some generalization. The novelty presented here of showing sequence number vs. time plots annotated with information gleaned by parsing the application protocol's requests and responses could be turned into a general, mostly table driven tool (though some protocols, such as HTTP, make this more ugly than others).

The tools we built for X are an initial foray into the art, and could clearly be greatly further improved. In particular netplot could easily be extended to display the statistics of selected regions of the graph, avoiding manual tallying of areas of interest when diagnosing non-startup related performance problems. We will probably add this capability sometime soon.

We have only explored a few applications, and while behavior is generally similar, there are major differences observed between toolkits.

In this paper, we have primarily looked at the startup phase of applications. Performance issues come up elsewhere. For example, drag and drop implementations

must interoperate and as of this writing also have performance issues. We can capture the entire application's execution, but we have not studied these other performance problems. Some work to make exploring much larger datasets might be fruitful, and tools for starting and stopping capture around interesting events would also be useful.

X is generally always faster than a 100megabit/second network in the tests we performed. Exploring behavior closer to on-machine performance would be aided by tests over a gigabit/second network; note, however, that there is useful parallelism between client and server. Our previous experience indicates that sometimes two machines running X over a network end up being faster than one; whether this would still be true for X today awaits further experimentation.

## 9 Availability

The tools used to capture packet traces and analyze them are all freely available under an MIT license in the author's public CVS server at <http://keithp.com/cvs.html> in the 'xcap' and 'netplot' modules. The source to this document can be found in the 'Usenix2003' module.

## References

- [AS90] Paul J. Asente and Ralph R. Swick. *X Window System Toolkit*. Digital Press, 1990.
- [BKO<sup>+</sup>02] David Boswell, Brian King, Ian Oeschger, Pete Collins, and Eric Murphy. *Creating Applications with Mozilla*. O'Reilly & Associates, Inc., 2002.
- [BS01] Daniel J. Barrett and Richard Silverman. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly & Associates, Inc., 2001.
- [Dal01] Matthias Kalle Dalheimer. *Programming with Qt*. O'Reilly & Associates, Inc., second edition, May 2001.
- [FK93] Jim Fulton and Chris Kent Kantarjiev. An Update on Low Bandwidth X (LBX): A Standard For X and Serial Lines. In *Proceedings of the Seventh Annual X Technical Conference*, pages 251–266, Boston, MA, January 1993. MIT X Consortium.
- [Gai93] Jean-Loup Gailly. *Gzip: The Data Compression Program*. iUniverse.com, 1.2.4 edition, 1993.
- [Get02] James Gettys. The Future is Coming, Where the X Window System Should Go. In *FREENIX Track, 2002 Usenix Annual Technical Conference*, Monterey, CA, June 2002. USENIX.

- [Gro00] NIST Internetworking Technology Group. NISTNet network emulation package. <http://www.antd.nist.gov/itg/nistnet/>, June 2000.
- [Har99] Eric Harlow. *Developing Linux Applications with GTK+ and GDK*. MacMillan Publishing Company, 1999.
- [MS01] Bart Massey and Jamey Sharp. XCB: An X protocol c binding. In *XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.
- [NGBS<sup>+</sup>97] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. Network performance effects of http/1.1, css1, and png. In *ACM SIGCOMM '97 Conference Proceedings*. Association for Computing Machinery, September 1997.
- [Pac94] Keith Packard. Design and Implementation of LBX: An Experiment Based Standard. In *Proceedings of the Eighth Annual X Technical Conference*, pages 121–133, Boston, MA, January 1994. X Consortium.
- [Pac01] Keith Packard. Design and Implementation of the X Rendering Extension. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001. USENIX.
- [Pin03] Gian Filippo Pinzari. The nx x protocol compressor. *Electronic Communication*, March 2003.
- [SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.
- [She90] Timothy J. Shepard. Tcp packet trace analysis. Master's thesis, Massachusetts Institute of Technology, 1990. Also *MIT LCS Tech Report 494*.

# Building a Wireless Community Network in the Netherlands

Rudi van Drunen  
*WirelessLeiden Foundation*  
*Leiden, the Netherlands*

rudi@wirelessleiden.nl

<http://www.wirelessleiden.nl>

Jasper Koolhaas  
*WirelessLeiden Foundation*  
jasper@wirelessleiden.nl

Dirk-Willem van Gulik  
*WirelessLeiden Foundation*  
dirkx@wirelessleiden.nl

Huub Schuurmans  
*WirelessLeiden Foundation*  
huub@wirelessleiden.nl

Marten Vijn  
*WirelessLeiden Foundation*  
marten@wirelessleiden.nl

## Abstract

With the development of low cost hardware based on IEEE 802.11b, wireless networks are an emerging technology. Using these wireless techniques outdoors it is possible to build a community network not dependent on any provider. In the Netherlands such a network is being set up in and around Leiden. Using low cost network interfaces, home-built antennas and open source software the volunteers of the Wireless Leiden Foundation were able to lay out an infrastructure for the inhabitants of Leiden at a very low cost. All kinds of applications (for-profit and not-for-profit) are using this wireless network.

## 1 Introduction

Current computer networks generally rely on a permanent, fixed and largely wired infrastructure which is owned and often operated by large entities such as telecom operators. A relatively new and emerging technology is wireless Ethernet, or wireless networking using the IEEE 802.11 standard. This standard encompasses the lower layers of the OSI model for transport of data as Ethernet frames using a spread spectrum based radio link.

This opens the possibility of building a network without having the problems and the cost associated with putting some sort of physical transmission medium in the ground. Instead, antennas can be used to send and receive the data using radio waves through free air.

Because of the relative simplicity of the currently available commodity hardware that uses 802.11 technology, it is relatively easy to build a local wireless community network in a town. Using this network people can share resources with each other. Examples are sharing sound or video files with the local museums, or hav-

ing data (text, images, video, audio files) provided by the local government on-line. Furthermore, the network can connect to the Internet providing a low cost way of crossing the last mile to the user at high bandwidth.

In Leiden, the Netherlands, a foundation has been established by a number of knowledgeable volunteers with the intention to build a network operated and owned by a community of users, not by big entities such as telcos or Internet Service Providers (ISPs). This essentially free network infrastructure can be used by anyone present in the service area for running his or her own application. On the client platform only an industry standard IEEE 802.11b interface and a probably small antenna is needed. Usage of the infrastructure is not "another monthly bill", but will be free, after a one time up-front investment in equipment.

## 2 Method

### 2.1 Introduction

The wireless community network built has to meet a number of requirements to be successful. First of all it has to be as "open" as possible to the users and to the developers. Being "open" enables anyone within the community to actively use the network and participate in the building thereof. Another constraint is that the network should be both reliable and low-cost. These constraints are met in this design using commercial off the shelf (COTS) and home built (low cost) hardware (network boards, antennas and PC hardware) components and Open Source software (such as Linux, FreeBSD and other packages well known to the Free Software community).

## 2.2 Technologies

A number of different technologies are available to build wireless computer networks. Most commercial solutions are proprietary to certain vendors or do not use low cost hardware.

The IEEE 802.11 standard (called WiFi in the commercial world) allows users to network their machines using radio technology. Different sub-standards have been formed specifying the bandwidth or radio frequency the networks operate on. The standard defines a number of operation modes which allow for ad-hoc, point to point and point to multi point networking. Though the standard calls for two transmission technology standards: Direct Spread Spectrum (DSS) and Frequency Hopping Spread Spectrum (FHSS), the market has by and large standardized on DSS for indoor and outdoor point to multi-point use. FHSS, which is more robust against certain types of interference and densely packed endpoints, is currently only seen on long point to point connections where interference and frequency allocations are at a premium, for example, at an aggregation point.

In this project we have chosen the 802.11b as primary standard mostly because of the availability of equipment, open source drivers and the cost of the hardware. IEEE 802.11b is a DSS radio technology supporting link speeds of 1,2,5.5 and 11 Mbit/s. The standard uses the 2.4 GHz frequency band and is initially designed for home and office use but with special measures (antennas) it is also applicable to crossing longer distances outdoors (up to a maximum of approximately 15 Km line of sight). [WirelessNet, WirelessComm]

## 2.3 Topology

The network we are building and operating is targeted to a coverage area of  $25\text{Km}^2$ , which is the complete city of Leiden (The Netherlands) and its surroundings. There are no hills in the target area. Other natural obstacles such as large forests are also absent. It is a small old town center with some moderately high buildings and some 10-15 story apartment buildings in the suburbs. These apartment flats are surrounded by small houses (max. height 3 floors).

The total number of people living in this area is approximately 160,000. In this area we want to provide outdoor coverage. For using the network indoors, a small antenna connected to the client computer has to be sufficient. The antenna should preferably have a line of sight to the nearest network access point. This is required as the maximum distance between the client and the network access point (a network node) is limited due to the national legislation implementing European (EU/ERC) regulations (restricting maximum radio frequency output power and restricting antenna gain).

Combining this knowledge with the anticipated traffic and bandwidth needs, it is evident that we need multiple nodes distributed over the coverage area. The nodes themselves have to be interconnected. A plot of the radio coverage of the current set-up (the historical center of Leiden ( $10\text{Km}^2$ )) is shown in figure 1. To provide full outdoor coverage of the target area an estimate of 25 network nodes will be needed. The interconnection



Figure 1: Current Wireless Leiden radio coverage plot. The total area shown is approx.  $20\text{Km}^2$ . Overlaid in black are contours of equal field strength of the areas in which the Wireless Leiden network is available when using a standard wireless network card connected to a 7dBi gain antenna located outdoors. (data courtesy of [www.wirelessdesign.nl](http://www.wirelessdesign.nl))

of the nodes also uses wireless links, making the network completely independent of the local (wired) infrastructure and thus very cost-effective and without significant monthly or other regularly repeating costs.

A mesh between the nodes will be formed, as each node is connected to the other nodes by at least 2 different (wireless) connections. With this approach, adding extra nodes to the network will add redundant paths and will therefore also increase the total available bandwidth. The topology as seen from the user is comparable to cellular telephony. Cells for users are created. In these cells the users share the total available bandwidth of an access point. The cells themselves are interconnected by point-to-point wireless connections. These connections form the backbone of the network.

## 2.4 Radio Planning

In the Netherlands there are 13 radio channels allowed in the 2.4 GHz frequency band to be used for radio local area networks. Due to the DSS technology a used channel does not occupy a single discrete frequency but is a distribution of power in a 22 MHz wide frequency interval. In figure 2 it is shown that there are 3 completely separate channels available. Combining this knowledge



with the topology and the goal of providing coverage in a fairly big area poses a challenging problem [Beckmann]. This problem is solved by careful selection of the channels to be used. For example, at the cost of a small increase in noise but no decrease in bandwidth, it is possible to use 4 channels on the same site. (e.g. channels 1,5,9 and 13). Of course the channel use is very much dependent on the local situation at the radio-level. Other measures to prevent cross-interference of different radio-links include the use of (directional) antennas and the location and polarization of the antennas. In the

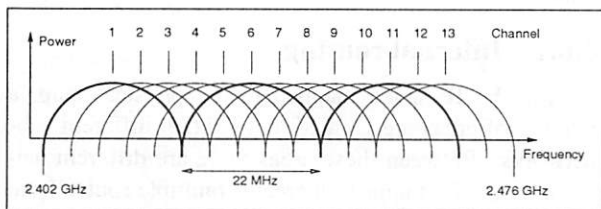


Figure 2: The available channels in the Netherlands.

Netherlands the 2.4 GHz frequency band is reserved for a number of licensed and unlicensed applications ranging from microwaves to video-links, vehicle identification systems and radio amateur (ham) use. The use of WiFi in this so-called ISM (Industrial, Scientific, Medical) band is allowed unlicensed, providing the above mentioned channel restrictions are observed and the effective output power of the antenna (EIRP) does not exceed 20 dBm (= 100 mW).

Providing coverage in the target area requires careful radio planning. Often chaotic behaviour is seen (i.e. small changes in initial conditions can turn out to be critical). Techniques such as hexagon planning, as used in the cellular telephone world, are used to optimize the coverage and spectrum (channel) efficiency. A homogeneous network is created by placing every site on a predetermined grid point such that every cell covers half of the inter cell distance. The physical cell boundaries are determined by the local situation and the antennas used. The cell boundaries are not symmetrical for up and down link due to local and receiver noise. This also needs to be taken in account to avoid areas without coverage (i.e. downlink possible, but no uplink possible). Using commercial Hata-Okumura [Ho] model based, radio-planning software we can simulate the propagation of the signal and optimize the location of the different sites. These advanced planning tools (e.g. CellCad ([www.lcc.com](http://www.lcc.com)) or PathLoss ([www.pathloss.com](http://www.pathloss.com))) are not comparable to those available in Open Source. Alternatives such as Radio Mobile ([www.cplus.org/rmw](http://www.cplus.org/rmw)) lack important data on the environment such as building characteristics. As most of the commercial tools require

a large computing infrastructure and various subscriptions to maps and other GIS data, like building heights and absorbency or reflection characteristics, it is not feasible to actually run the simulations ourselves. A sponsor has access to this software (CellCad) and runs the required simulations.

Nevertheless, a site survey is always needed to actually measure the noise generated by other radio sources on-site and check the signal strength of the already running nodes. Due to the high absorbency of the radio signals by (for example) trees in the line of sight, local field measurements are essential in planning a node. A site survey is done using the Kismet [Kismet] or dstumbler [Dstumbler] software running on a Linux or FreeBSD laptop with a small panel or directional antenna and wireless network interface.

As a subproject within the community a compact automatic survey tool is currently under design. This tool will collect signal strength and noise figures from existing Wireless Leiden nodes as well as signals from access points or other devices in the 2.4 GHz band. The hardware device based on an embedded system connected to a GPS receiver can be carried through the city on various utility vehicles to effectively cover the complete area. Once completed, the software environment will be comparable to the software setup of a node with some extra tools to log the data. When available, the logging data will then be plotted as an overlay on the simulation maps and made available to the users wanting to connect to the network as a guideline for aiming their antennas.

## 2.5 Antennas

The standard antennas that come with wireless interface cards are designed for office use. To use this network interface in a long distance outdoor connection as we do, different (external) antennas are needed.

### 2.5.1 Antenna types

Good access point antennas are omni directional or sector types. An omni directional antenna has the advantage of quickly providing a large coverage area. The usage of sector antennas makes it possible to split the area around a node into several different independent parts. The main advantages of this approach are that while providing a stronger signal into the sectors, the access point node is able to handle more users (i.e. a larger total available bandwidth for that cell). A disadvantage of using sector antennas is that more wireless interfaces are needed and that interference can be a problem.

A point-to-point connection needs an antenna which directs all the radio frequency (RF) energy into one direction. The limiting factor for this antenna is mainly its size. High gain antennas reduce the chance of interference to other nodes or to the other antennas of the same

node but are often quite large in size.

The key to a successful antenna setup for a given node location is in evaluating the simulation results together with the site survey details. After that, the cost factor is not to be neglected in the selection process. In some cases a home-made antenna can have a better price / performance ratio than a commercial one.

Clients need a directional antenna to connect to a node. There is no need for clients to be connected to more than one node at the same time. The main specification for the client antenna is the minimum gain that is needed to obtain a signal strong enough to make a good connection to an access point. Based on this gain, several different types of antennas are available. Other factors that determine the best antenna are the physical size, appearance and the cost of the antenna.

Simple antennas for use with access points [Omni], clients [Quad] or point-to-point connections [Yagi] can be made at home, keeping the price at the lowest possible level. Almost anything can be turned into an antenna. It has been shown that it is not difficult to make a WiFi antenna [Pringle]. The Wireless Leiden website has a large collection of antenna designs provided by users.

The layout of the Wireless Leiden network is designed on a typical client antenna gain of 7 dBi. Antennas with this gain are cheap and / or easy to construct.

## 2.5.2 Lightning protection

Lightning protection is only applied at the building level, like connection of the antenna pole to the existing lightning protection system. This means that there are no special surge protectors used in the antenna cables. The chance that lightning strikes in the Netherlands is so small compared to the cost of the arrestors and the problems associated with them (like insertion loss and the protection level they provide) that the risk of getting the (low cost) equipment damaged by lightning is acceptable.

## 2.6 IP space Planning

The network uses TCP/IP as the transport layer. Therefore, every active element in the network should have an address. Using a private IP version 4 range, enough addresses will be available. Using IP version 6 will be a future enhancement and is not yet fully implemented. So, as the IP network grows there is a need for not only planning the radio frequency space, but also planning the IP space. The IP range is assigned on the basis of the different postal (zip) code regions in the coverage area, combined with the population density and average income per head. Client machines are provided with the correct network information for the area they are in by a local DHCP server running on the (first) node (the access point) they are connecting to. The information provided

by the DHCP server is the IP address to use, the netmask, the nameserver addresses and the default gateway.

Any two nodes in the backbone that are connected to one another by a point-to-point link use a /30 IP range (i.e. 4 IP addresses). This ensures that traceroute shows the logical network topology (no physical hops are missing because all packets must rise to the IP level, none are routed by MAC address). IP space that is used for numbering an interlink from node A to node B is assigned from a separate IP range; it should belong to neither geographical area A or B, as it is part of the backbone.

### 2.6.1 Internal routing

Having the IP space mapped to the physical map, a number of areas are created that contain different sub-networks. Between these areas there are different network links. This approach creates multiple routes from a source to a destination. It is no longer possible to manually manage the routing tables in all nodes that have more than one interconnect. Using the Open Shortest Path First (OSPF) routing protocol minimizes the routing configuration of a node. As soon as the appropriate interlinks are made (or broken) on the IP level, the OSPF processes start to exchange information about their knowledge of the network topology. This also allows us to add or remove nodes from the network without any routing reconfiguration.

Using the hop count and cost factors on various routes, the system will select the route with the lowest total cost. If any link fails, it is detected by the OSPF daemons and all routing tables in the network will be updated to reflect the new situation. Currently, routing is done solely on hop count. All cost factors are the same.

By using OSPF routing and different interconnects on one node, the reliability of the network on the IP level is greatly enhanced at a negligible cost of processor overhead and network bandwidth.

For the implementation of the OSPF protocol the Zebra [Zebra, Routing] package is used. Zebra is an Open Source package which provides a number of different routing protocols by various daemons, all controlled by a master daemon. Currently OSPFv2 routing is used throughout the backbone network because the protocol is fit to the topology and does not impose a large overhead.

### 2.6.2 External routing

Using a private IP space on the network will introduce new issues on routing when connecting the network to the Internet using multiple ISP connections. Some nodes have an external default gateway that is injected into OSPF. These routes are propagated through the network so that each node has one or more routes to the Internet.

Currently there is no smart algorithm to decide which one to use (or to use a particular route to the Internet for a particular user). Several options are being investigated: tunnels (e.g. PPTP), source routing and the use of proxy servers.

Also, discussions with RIPE are planned to address the possibility of obtaining a block of IPv4 numbers together with an AS number. This will ease some of the ISP connection problems.

## 2.7 Site allocation

Once a site is designated by using the planning procedure, it might be a tedious task to get permission from the different building-owners to have the antennas and equipment installed. Because of the not-for-profit and volunteer-driven nature of this project, it might be difficult to explain to the building owner that we are not able to pay the same amount of money as a cellular phone provider. A cellular provider will easily pay up to Euro 10,000 each month for a location. We, as Wireless Leiden affiliated volunteers, have organized ourselves in an official charitable trust (foundation) with a statute, and this foundation has managed to generate some positive publicity. Being an official foundation with some media exposure and a good story – a free, fast community network – has been helpful in gaining rooftop access free of charge. Having access to people knowledgeable on building and safety issues ensures a setup according to all local building regulations, like measures against lightning strikes.

Locations that are important to the community, like schools or the Town Hall, are target locations to set up nodes. Also cooperating commercial enterprises that want to provide services on the network or want to make use of the network for private communications (e.g. between different branch offices) are quite willing to invest in the equipment and time to set up and maintain a node.

## 2.8 Setting up a Node

A typical network node setup consists of a number of antennas and a computer system. We use 2 or more directional antennas to connect to other backbone nodes and one omni-directional antenna for local client access (See figure 3). A PC or other system provides the routing and access-point functionality. A typical node setup can be found in figure 4. The home-built and partly commercial antennas are connected to a computing platform that can, but does not need to be connected to the local network at the site using wired Ethernet. Setting up a network node requires some real hard hardware work to get the antennas lined up and affixed to the building. As we are using directional and polarized antennas to prevent interference, the alignment of the antennas is fairly critical. Once the antennas are set up, connection to the node

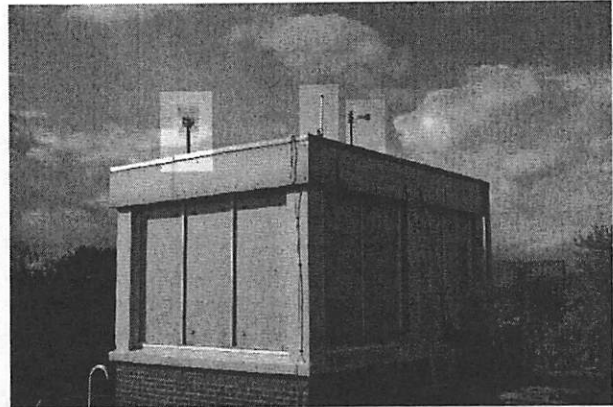


Figure 3: Antennas as mounted on a building (2 directional antennas and one omnidirectional antenna).

machine is done using low loss coaxial cable. As keeping cable losses to a minimum is important, the length of the cable should be minimized as each meter of antenna cable introduces a loss of approximately 0.25 dB. The node machine should be as close as possible to the antennas. When using a small embedded system (see section 2.9) the node computer can even be mounted outside on the antenna pole. In this case, special measures have to be taken to weatherproof the setup (e.g. like preventing condensation). This setup can use Power over Ethernet (PoE) [PowerF] [PowerS]. A single UTP network cable (where the length is not an issue provided that it is less than 100 meters) connects the antenna setup to the power supply (indoors) and the on-site local wired Ethernet if needed.

Before and after the installation of the node hardware a number of basic alignment, throughput, and reliability tests have to be run. These tests are used to be sure that the links in the set-up will operate as expected. The expectation values for these tests are derived from the site survey and simulation results as described in section 2.4. The alignment test comprises of monitoring the signal strength and noise figure when a backbone link is established. By physically varying the antenna direction the signal as seen by the driver software on the wireless interface should be maximized, while the reported noise is minimized. Throughput and reliability is tested by transferring large chunks of data from one node to its neighbor and observing the packet loss and transfer rate.

Often, once the node is set up it is quite difficult to physically access the machine and the antennas because they mostly are located at remote locations and in buildings with complex access procedures. A typical test for the reliability is to copy some video data (often comprised of large files) through the node. This test differs from the test mentioned above where a single wireless link is tested, here the complete (routing) functionality



of a node is tested. A test protocol is used to assure the repeatability of the test procedures.

An external hardware watchdog device is used to ensure a reboot of the system when some part of the software crashes. These watchdog devices are home built and very simple. The watchdog watches the serial port of the node computer for a "hello" string. This string has to be sent every minute (using a small program or even a script). If this string is not received within 90 seconds of the previous string, the watchdog circuit will cut the power to the setup and will turn it on after a minute. Using either journaling file systems or RAM based file systems minimizes problems caused by a fsck operation after the setup has been powercycled.

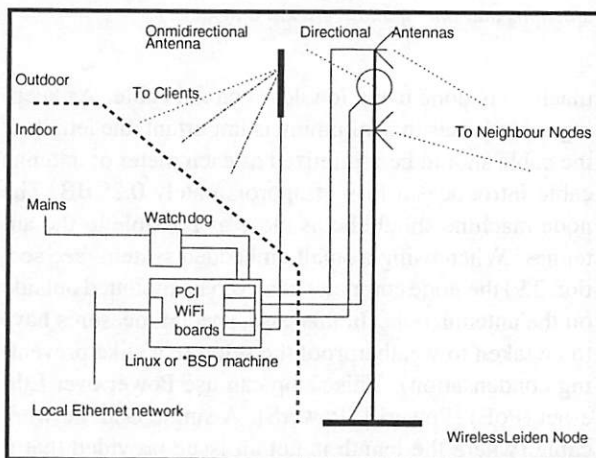


Figure 4: A typical WirelessLeiden Node setup.

The software environment on the machines is a standard Open Source free UNIX (currently FreeBSD) distribution stripped down to fit in a minimal hardware configuration. The (kernel) device drivers that are used to control the wireless network cards and some network operations and management utilities are added (ref. section 2.10).

Using a standard off-the-shelf Open Source operating system enables us to implement a node quite fast while at the same time keeping the flexibility of changing things on all levels when the network grows and / or the technology changes. Another aspect is the large and diverse knowledge-base available within the development and engineering group. Also, the Open Source development model guarantees a fast turn around time in fixing bugs or evaluating features. Last but not least, in a not-for-profit organization working from donations the initial cost of the software is of major importance.

## 2.9 Hardware Platform

Each node needs a piece of hardware that handles the wireless signal and provides IP routing. Currently Intel

Pentium I class PC machines with a PCI bus are used. The wireless interface cards are standard cards using the Intersil PRISM series chip set and a PCI bus, such as the Linksys WMP11 or Compaq WL200. The PCs used are often found as surplus machines or are donated. Due to the different hardware configurations of the machines it is difficult to standardize the hardware and improve the reliability. To overcome these problems different options are evaluated. Commercial wireless nodes (e.g. the Nokia rooftop systems) are not an option as they are often not "open" and far too expensive. Often they use proprietary adaptations from the 802.11 standard and are not accessible to the developers in the community to develop the hard- and software further.

Embedded systems based on i486 or better processors with at least one Ethernet interface and at least 2 slots for connecting wireless network cards are a good alternative. When using an Intel i486 or better processor most of all software developments can be reused.

Embedded boards are available in various different flavors. At this moment we are testing the Soekris Engineering embedded boards. These boards use an i486 compatible processor (AMD Elan SC 520) running at 133 MHz, 64 Mbytes of RAM, a Compact Flash slot, a mini PCI slot and two PCMCIA slots. Figure 5 shows a block diagram of this board when used as wireless network node. For the PCMCIA and Mini PCI solution wireless cards based on the PRISM 2 reference design (e.g. SENA0) are used. This adds software compatibility with the PRISM 2.5 based PCI cards that are used in the PC design. The Soekris board is a commercial product and will run either Linux or a flavor of BSD without problems, freeing the developments in the node software from the hardware developments. Other industrial Intel based boards often use various bus standards (e.g. PC 104), which add additional costs in connecting (standard) cards for wireless networking. Self-designing a board level solution is not an option due to lack of resources and money. At a price higher than the price of the donated PC machines these embedded systems will add reliability and standardization to the node base. The intention is to migrate the core network nodes from PC's to embedded systems.

## 2.10 Software Platform

The main choice of Operating system for a Wireless Leiden Node is FreeBSD. In the development of the Wireless Leiden node a number of problems were encountered which have resulted in the current FreeBSD installation.

Initial Linux implementations of the node functionality resulted in problems with the performance and stability of the systems. Partly they were due to the older hardware used: the PCI controller not allowing to assign



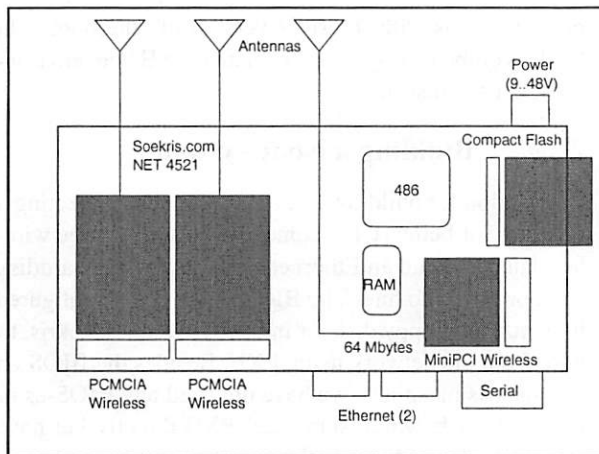


Figure 5: A Soekris based node.

different interrupts to the wireless cards and the interrupt handling in the Linux kernel. Other problems were connected with using multiple wireless cards with the PRISM hostap driver [HostAP]. We could have used other drivers (e.g. wlan-ng), but the hostap driver was the only one implementing an user software (i.e. not firmware) driven access point functionality. Another major issue was the number of different Linux distributions around and the rapid change thereof.

Using FreeBSD solved some of the Linux-related problems, like interrupt handling on heavily loaded systems, while adding features such as having one build tree, a steady release and distribution scheme. Furthermore, a single stable PRISM 2 support stack and access to a more mature configuration system makes the system more suitable for mass deployment. Another benefit of using FreeBSD is the inclusion of several wireless network card firmware cutoff checks which will restrict the functionality accessible from userland when a too old version of the firmware is detected.

For the Compaq WL200 card we did have to fix a specific routing bug which was introduced in between FreeBSD 4.6 and 4.7 as part of the cardbus / newbus project. This has since been picked up by CURRENT and will be in the next release of FreeBSD. Prior to FreeBSD 5.0 release some issues were also found and reported with regard to the wicontrol and ifconfig settings not being propagated properly. These have since been fixed.

It should be noted that each of the different wireless stacks on Linux has at least one or two elements which are vastly superior to the rather dated PRISM 2 support in FreeBSD; but unfortunately each stack also exhibited showstoppers ranging from erroneous detection of cards to kernel panics. It is not unlikely that a maturing or crystallizing, of one of the wireless options of Linux will make Linux a viable choice in the near future. This,

combined for example with a good AODV routing protocol implementation (see section 3.4), may again cause us to switch platforms. Another, unplanned, benefit of the BSD stack is the rather mature diskless boot system which is part of the standard distribution; which is an excellent starting point for automated installers and systems operating from a read-only storage device.

As FreeBSD has just a single version number and release path; the entire 'build' procedure of the master machine from which all nodes are automatically built entails less than 5 kbytes of script; just short of one page. The 'definition' of a node, including kernel configuration, routing, antenna frequencies etc.; i.e. all that needs to be defined given a specific version of FreeBSD, currently runs at around 12 kbytes of data.

Tools such as 'mergemaster' available in FreeBSD and the rather elaborate documentation make upgrades in the field, even across versions, reliable and predictable. Likewise extensive use is made of the 'ports' collection to manage the versioning and updating of "third party" packages such as the ISC-Bind, Zebra and a few others.

## 2.11 Version control

Deploying a large number of network nodes consistently and reliably, depends for a large part on the version control of the different configurations and their configuration files. In the Wireless Leiden project Subversion is used together with a system called Genesis.

Subversion [svn] is a new breed of CTM; akin to SCCS or CVS - but more suited to Open Source code management as it does not require extensive Unix account management and can be configured to use HTTP as its communication protocol. Currently, Subversion is hosted on a remote machine and is connected to the Internet through a DAV module and an Apache 2.0 web server. Access controls are intentionally light and commit access is virtually for the asking. Genesis is a web based homebuilt configuration file generator written in PERL [Perl5].

Due to the remote installation of both the Subversion and Genesis installation and configuration of a Node machine relies on a live Internet connection. However, it should be noted that both systems could be moved to a more 'local' environment close by or could be replicated. So far this has not been necessary.

## 2.12 Building the Nodes

Apart from doing the real hard hardware setup of a node, the software configuration on the system has to be set up. To automate the software installation and configuration, the Wireless Leiden Node Factory was developed. A non trivial solution was needed because node configurations differ more than just by their IP address, due to different

hardware configurations and platforms being used.

### 2.12.1 The Node Factory setup

The node factory consists of a central machine which contains the distribution(s) to be installed on the node along with a diskless boot environment. The latter is used during the initial boot as a staging ground for the actual install. A new machine is connected to the central machine, booted from floppy using 'etherboot' which subsequently launches into a diskless FreeBSD install. All hardware will be detected, a reformat of the hard disk is done and the node software is installed and configured.

The install master server is a machine with two network cards and a single wireless network card. At present a Pentium II class machine with 256 Mb memory and 4 Gb harddisk is used.

The first network card connects to the Internet through a DSL connection. This is needed to fetch source, binaries and configuration files. The second network card is connected to the machine to be installed and configured. The wireless network card is used for the automated tests during the installation.

As the first network interface on the server provides Internet connectivity, the second nic offers a NFS, an etherboot and a PXEboot environment to the client to be installed. To enable NFS standard FreeBSD `rc.conf` and export settings are used; etherboot floppies are created using the default version in `/usr/ports` and the PXEboot environment relies on the DHCP daemon using a standard configuration from the handbook.

In order to allow a newly installed node to access the Internet routing is enabled between the nics by a `gateway_enable` setting in `rc.conf`.

The software is retrieved from the Net by downloading a FreeBSD image. After installing this on the server software was upgraded from 4.7 to the CURRENT branch of FreeBSD and frozen. The CURRENT branch was selected over the STABLE branch at that time to make sure the latest wireless drivers and patches thereof were included. However, testing on basic functionality was needed to validate the distribution for production use.

Using the `make world` mechanism with a different `DESTDIR`, a complete separate FreeBSD tree is built. Added to this tree at the root is a special diskless install kernel and, in the default `/boot/kernel` location, a stripped down run time kernel. Furthermore, two additional scripts are added in the root which will control the actual install. An extra `rc.local` script is added which will run as the final step during the first diskless boot. The latter causes the disk to be partitioned, formatted and the install to start.

The next step is making a bootfloppy for each of the

network cards with the right version of etherboot. On Soekris embedded systems the native PXE boot environment is used instead.

### 2.12.2 Building a Node system

Preparation for building a new nodes means collecting a Pentium (or better) CPU some RAM, two or three wireless lan cards and an Ethernet card. A 500 Mb harddisk (or more) will do fine. The BIOS of the PC is configured to boot from floppy disk, or in the case of the Soekris, to boot from the network using PXE. Besides the BIOS on the Soekris machines, we have not tried any BIOS-es or Ethernet cards which supported PXE directly but have relied on a few different etherboot floppies instead.

In the first stage the new machine boots from floppy disk to initiate etherboot. DHCP will assign an IP address and provide the parameters which allow the node to mount the NFS file system. After loading the kernel, a script prepares the harddisk by making it bootable, defines its partitions and formats file systems. This is followed by copying all the files from the server to its own harddisk. After a reboot the new machine has its own system in order to run.

After the reboot installation of some packages such as a `dhcp-server`, a `nameserver`, `snmp` etc. takes place. The installation is managed by a set of homebuilt scripts (see section 7). The reason to do so after a reboot rather than during the diskless boot is that during the initial diskless boot we do not want to rely on having sufficient memory available. The diskless system uses a memory based `/var` and `/tmp` overlay and therefore consumes quite some memory. The scripts are invoked by a state engine at the end of the boot procedure.

At the end of the procedure the new machine will show a list of hostnames. After entering one at the command line the machine will lookup its configuration on the Internet in the remote Genesis. The old files will be backed up and the new ones that define the specific node will be put in the right places.

Then approximately 15 minutes after the first (floppy) boot, the new machine is in operational state and ready to be deployed and located at site.

## 2.13 Design rationale

Using this Node Factory setup for building a network node it is very easy to setup and deploy wireless nodes, while ensuring a maximum of maintainability. The low-level configuration of each node is nearly identical and by using version control and the configuration database it is very easy to track down changes and keep these documented. A last important point is that using Open Source software the complete system will end up to be very cheap.

## 2.14 Security

Security is currently not applied on the network infrastructure level. Of course, all network nodes have appropriate security to secure the boxes themselves, but as an infrastructure provider, we have the rule of "security is the responsibility of the user". On the radio level we use a combination of narrow beams (directional antennas) to interconnect nodes together with Wired Equivalent Privacy (WEP) or even a WEP infrastructure with dynamic keying. As WEP provides no actual security [Borisov], the user of the infrastructure must be aware of the *insecurity* of the transported data, and use security on a higher level, for example by using IPsec tunnels over the existing infrastructure. Raising awareness of these problems and their solutions by the users is important. Right now the projects website addresses this in detail.

The nature of the transport layer adds an extra possibility of Denial of Service (DoS) attacks by "jamming" a connection on the radio level. This can happen when another device using the same frequency is operating in close vicinity of the node. Because we are operating concurrently with other users in the same frequency space this can be a problem. Adding redundant paths together with the appropriate routing protocols is the way to overcome the problem for the user.

## 2.15 Running a Network Node

The network node is highly self-contained and does not need frequent maintenance visits. All software and configuration maintenance, upgrading etc. is done from the network itself using the configurations in the repository. Software level reconfiguring of the node can be done on the fly to cope with the changing network architecture.

However, some on-site hardware maintenance may be needed. Typical issues to be considered are taking care of the fans in the power supply and coping with hardware failures such as hard disk problems. Furthermore, some hardware maintenance on the antennas is needed. Regular inspections of the state of the antennas and mounting hardware are required to ensure safe and reliable operation. Local building safety regulations may also require regular on-site inspections of the set-up.

The Network node can be connected to the users' (home) network in order to give the local user wired access to it. Therefore, on the Ethernet port DHCP is enabled.

Traffic and other operational data on the node is gathered using RRDB and RRDtool [RRDB] and sent to the central repository at regular intervals. Some examples of graphs used can be found in figure 6. For simple "health" management of the nodes the free version of Big Brother [BB] is used. Big Brother generates a number of "status lamps" on the network map available on

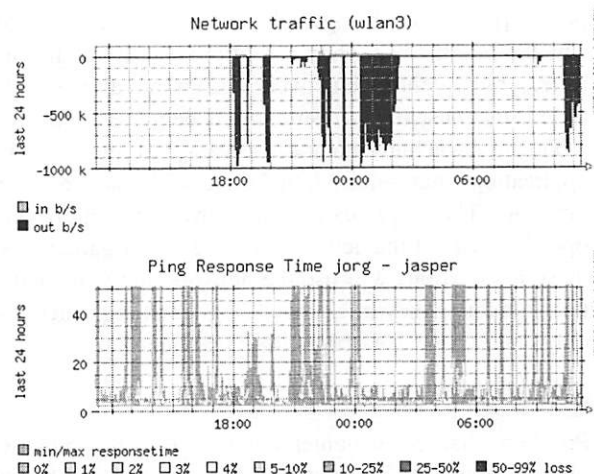


Figure 6: A link traffic and ping latency graph made by RRDtool.

the website. The combination of the graphs created by RRDtool together with the *ok-notok* information from Big Brother has shown to be a valuable tool to monitor the network. If the network and its traffic grows, other tools may be needed.

## 3 Results

Within approximately 1.5 years the network has grown from a single point-to-point connection to a completely usable network. Logging MAC address data shows that the network is used by approximately 300 different machines. On a day-to-day basis it is currently estimated that approximately 100 users are connected. With the introduction of (free, sponsored) Internet connectivity on the network this will without doubt grow. Although the learning curve was steep, the fully operational network is now gaining more acceptance as more applications become available.

### 3.1 Applications

During the first year of this project we have succeeded in building about 12 network nodes in the Leiden area. These nodes are interlinked with their neighbors by 802.11b links and have access points to accommodate users to connect wireless to the network. Applications currently in use are various VPN connections of enterprises giving their employees access to the company network, schools with different locations connected to each other, a video server and some gaming applications.

As of March 2003 a large Internet provider is sponsoring the Wireless Leiden Foundation with a number of high speed (8 Mbit/s) DSL connections. Without raising big routing and peering problems the first phase of integrating Internet connectivity to the Wireless Leiden network a number of proxy servers [Squid] and captive



portals [NoCat] are being deployed to allow the clients to surf the Internet using http and https only. In the meantime, a more definitive network design that allows multiple ISPs to use the infrastructure is being developed.

Due to the achieved outdoor coverage of the network, applications that require mobile use of the network are possible. These applications are currently being developed by users of the network. We are investigating how these applications will use the network and what additional features they might need (e.g. roaming functionality and / or the use of MobileIP [MobileIP]).

### 3.2 Problems

Problems that are encountered during the start-up phase of the network can be divided into two groups. There are technical and non-technical problems to be solved.

Technical problems are seen on every level of the network stack. Starting with the physical level, it is difficult to plan the network using the different constraints like noise, limited channels available, natural and other obstacles. Due to this issue the configuration of the network is constantly evolving. Changing of antenna directions can be difficult once a node is set up.

On the higher levels we have encountered problems with the network drivers for the wireless boards we are using. These problems can seriously affect the throughput and reliability of the network. Here the use of open source software and the open source development model pays off. Problems can be solved within the group, or with help and information gained from the Internet. Also having a choice between different solutions to the problem helps.

On the IP level, having many point-to-point links and a number of applications results in quite a big puzzle to get everything configured correctly. Using a central configuration repository and the use of routing protocols helps, but some problems still remain to be solved. For example when connecting to the Internet (with a number of providers), routing and numbering issues needs to be solved. Also the latency in the network might pose a problem in some near real-time applications or applications that use streaming technology (e.g. live video).

Furthermore, the reliability of the hardware can be a problem. Because of cost issues, complete (used) PC machines are in use as network nodes. These machines are not as reliable as dedicated routers running on embedded systems without moving parts (like fans or hard-disks). Minimizing the points of failure, testing, validating and proper maintenance helps a lot here. Ultimately, using embedded systems will ease this problem.

Non-technical problems include gaining acceptance within the town and the community. Without a broad acceptance and support of various organizations it is not possible to build this kind of a network in a not-for-profit

fashion. Access to rooftops of high buildings etc. is essential. However, we do not believe that a for-profit organization has any chance of succeeding at all. The upfront costs of building a network without volunteers are very large. Possible revenue streams will be insufficient to recoup those large costs.

The other difficult task is to manage a large group of volunteers that are actually doing the work. With the growth of the network, also new people are joining to do the design, management and the building of the network and the nodes. With every new engineer another degree of (intellectual) freedom is added to the system. Managing this enormously large and diverse task force in an open manner can be difficult.

Also, we see that on the non-technical level the more religious ideas between e.g. the different operating systems or engineering solutions on a specific topic are a problema to cope with (like \*BSD vs. Linux). Having a heterogeneous set-up with different operating systems will gain a broader acceptance within the group and probably builds a more robust network, but will be much more difficult to manage.

### 3.3 Experiences

Right now, the network is in full operation and the first applications are successfully being deployed. Due to the rather small group of technically skilled people the main focus of Wireless Leiden has been to set up nodes and provide coverage in the historical center of Leiden. In addition we have concentrated on knowledgeable individuals and professional organizations as first users, because they require less IT-assistance from the volunteers. As this is accomplished, the next step is extending the network to the suburbs and connecting the actual private users to the network.

Doing research in this environment is very attractive: the big advantage is that a testbed (i.e. an actual running network) is already deployed, so actual field testing of new technologies is relatively easy.

### 3.4 Future work

For the coming year a target is set to build and install at least one new network node every month to extend the coverage and increase reliability and bandwidth. The second target is to set up a structure to effectively help the individual users to connect to the network. A third target is getting more applications running.

New technologies are being tested to cope with the expected growth of the network. Upgrading the backbone links between network nodes to use 802.11a or 802.11g technology (max. 54 Mbit/s) [WCompare, IEEE] is in test.

Also in evaluation is the use of more complex and advanced technologies using mesh or ad-hoc network-



ing [Hu, Maltz, Royer]. Imagine a network where all clients are also a node and every client is connected to the two or three closest other clients (close is defined here as the distance at which one is able to create a *good* (signal to noise ratio) connection at that particular moment in time), not to a central node in the neighborhood. The noise floor would remain at the same level as the network gets more dense while the number of possible parallel paths from A to B also grows with the number of clients. More clients lead to shorter paths which will result in less RF power needed to obtain the same connection quality. Each new participant brings along his or her own piece of network, bandwidth and noise-reduction.

Using these techniques, advanced routing architectures such as AODV may be used. The Ad hoc On Demand Distance Vector (AODV) routing protocol [AODV, aodv-ietf] is an "on demand" routing algorithm, only creating routes when they are actually needed. AODV is loop-free and scales to large numbers of mobile nodes.

Another point of evaluation is the use of applications that require Quality of Service (QoS) facilities in the network such as IP telephony and wide band video streaming.

## 4 Related work

In different parts of the world wireless communities are developing. In the USA there are a number of leading initiatives [nyc, Seattlewireless, FreeNetworks], but also in Europe and Australia communities have been formed. The main difference between the Wireless Leiden network and a number of other wireless communities is that Wireless Leiden definitely not has a hobby-network kind of style and set-up. Due to the professionals affiliated with the Wireless Leiden Foundation and the partnerships with major (local) groups of potential users, hardware vendors, the university and content providers it is far beyond the concept of a number of people sharing their DSL connection using wireless technologies. The other main difference is that due to the acceptance by the community and therefore the possibility to set up nodes on a large number of non-individually owned buildings, all connections between the network nodes can be wireless. No wired connection is needed. Here the European (or Dutch) mentality of cooperation to achieve the best result may be an advantage.

Last but not least, as the infrastructure is based on open standards and Open Source software, it is freeing us from vendor lock-ins and is achieving the broadest possible range of applications while providing sustainability of the complete system.

## 5 Conclusion

With the use of relatively low cost technologies and Open Source software it is possible to build a wireless network which is used by the community. Technical problems do exist, but by usage of Open Source software they can easily be solved. The process of building a network using a loosely-coupled group of volunteers is not easy but bringing organization in the group when the network becomes more complex helps a lot. Having the back-up of the community and local enterprises also helps to gain momentum and visibility of the project, which in return speeds up the development and growth of the network.

## 6 Acknowledgments

The authors wish to thank all people affiliated with the Wireless Leiden foundation, The city of Leiden and the sponsors of the project. Without the dedication, professionalism and enthusiasm of these people and organizations this project would not be able to prove its successfulness. Special thanks go to the following people: Johan de Stigter of Gandalf and Evert Verduin of WirelessDesign for their technical input. Furthermore thanks go to Caroline Beijer, Brad Knowles and Dick van Velzen for their valuable comments. Special thanks also to the FreeNIX reviewers and to our shepherd, Guido van Rooij.

## 7 Availability

As the Wireless Leiden Foundation is an open community, all information, software and hardware that is being developed is free to use for everyone under the Gnu Public License. A WiKi website is available to effectively share this information. Unfortunately it is in Dutch, but as engineering language is international, with some effort the important technical info can be extracted. The website can be found at:

<http://www.WirelessLeiden.nl>

Direct links to the node software can be found here: <http://www.WirelessLeiden.nl/wcl/cgi-bin/moin.cgi/InstallLayers>  
The subversion tree is available under the following link: <http://wleiden.webweaving.org:8080/svn/node-config/master/>

## References

- [AODV] AODV: a routing protocol for ad hoc networks,  
[http://w3.antd.nist.gov/wctg/aodv\\_kernel/](http://w3.antd.nist.gov/wctg/aodv_kernel/)
- [aodv-ietf] Ad hoc On-Demand Distance Vector (AODV) Routing draft-ietf,

- <http://http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-13.txt>
- [BB] Big Brother: a network monitor,  
<http://www.bb4.com/>
- [Bawug] BaWUG, The Bay Area Wireless Users group,  
<http://www.bawug.org>
- [Beckmann] D. Beckmann, U. Killat, *Applied Frequency Planning for Cellular Radio Networks*, Int. Journal of Electronics and Communications, 54-4 2000 pg. 211-217, 2000.
- [Borisov] Nikita Borisov, Ian Goldberg and David Wagner, *Intercepting Mobile Communications: The Insecurity of 802.11*, 7th Annual International Conference on Mobile Computing and Networking, 2001.
- [Dstumbler] dstumbler: a wardriving / netstumbling / lanjacking utility for bsd operating systems,  
<http://www.dachb0den.com/projects/dstumbler.html>
- [FreeNetworks] Free Networks, information on different community networks,  
<http://www.freenetworks.org/>
- [HostAP] Host AP driver for Intersil Prism2/2.5/3,  
<http://hostap.epitest.fi/>
- [Hu] Y. Hu, A.Perrig, D.B.Johnson, *Ariadne, a Secure On-Demand Routing Protocol for Ad-Hoc Networks*, MobiCom 2002,(2002).
- [Ho] Hata-Okumura Model,  
[http://wcrgr.engr.ucf.edu/Web\\_Paper/HATA.html](http://wcrgr.engr.ucf.edu/Web_Paper/HATA.html)
- [IEEE] IEEE working group for wireless LANs,  
<http://grouper.ieee.org/groups/802/11/>
- [Kismet] Kismet sniffer,  
<http://www.kismetwireless.net/>
- [Maltz] David A. Maltz, Josh Broch, and David B. Johnson, *Experiences Designing and Building a Multi-Hop Wireless Ad Hoc Network Testbed*, MU School of Computer Science Technical Report CMU-CS-99-116. March 1999.
- [MobileIP] IP Routing for Wireless/Mobile Hosts (mobileip),  
<http://www.ietf.org/html.charters/mobileip-charter.html>
- [NoCat] The NoCat Community Wireless Network Project white paper,  
<http://nocat.net/nocatrfc.txt>
- [nyc] New York City Wireless,  
<http://nycwireless.net/>
- [Omni] Trevor Marshall, *802.11b WLAN Waveguide Antennas*,  
<http://trevormarshall.com/waveguides.htm>
- [Perl5] Perl5 Programmers reference,  
[http://www.metronet.com/perlinfo/doc,\(1996\)](http://www.metronet.com/perlinfo/doc,(1996)).
- [PowerF] Power over Ethernet FAQ,  
<http://www.nycwireless.net/poe/>
- [PowerS] Power over Ethernet standard working group  
<http://grouper.ieee.org/groups/802/3/af/>
- [Pringle] Rob Flickenger, *Antenna on the Cheap (er, Chip)*, O'Reilly weblogs, Jul 5 2001.  
<http://www.oreillynet.com/cs/weblog/view/wlg/448>
- [Quad] The Dutch double Quad antenna,  
<http://sharon.esrac.ele.tue.nl/pub/antenne/13-double-quad.jpg>
- [Routing] R.Malhotra, *IP Routing*, O'Reilly & Associates, Inc. (2002).
- [Royer] E. Royer and C.K. Toh, *A review of current routing protocols for ad-hoc mobile wireless networks*, IEEE Personal Communications Magazine, pp 46-55 (1999).
- [RRDB] Tobi Oetiker, *RRDB and RRDtool*,  
<http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/>
- [Seattlewireless] Seattlewireless, a wireless community in Seattle,  
<http://www.seattlewireless.net>
- [svn] Subversion,  
<http://www.tigris.org/>
- [Squid] Squid Web Proxy Cache,  
<http://www.squid-cache.org/>
- [WirelessComm] Rob Flickenger, *Building Wireless Community Networks*, O'Reilly & Associates, Inc. (2001).
- [WCompare] Netgear *Wireless Comparison Whitepaper*,  
[http://www.netgear.com/pdf\\_docs/WirelessInforev4.pdf](http://www.netgear.com/pdf_docs/WirelessInforev4.pdf)
- [WirelessNet] M. Gast, *802.11 Wireless Networks*, O'Reilly & Associates, Inc. (2002).
- [Yagi] SeattleWireless, *Building Yagi antennas*,  
<http://seattlewireless.net/index.cgi/BuildingYagiAntennas>
- [Zebra] The ZEBRA routing package,  
<http://www.zebra.org/>

# OpenCM: Early Experiences and Lessons Learned

Jonathan S. Shapiro     John Vanderburgh     Jack Lloyd  
shap@cs.jhu.edu     vandy@jhu.edu     lloyd@randombit.net

*Systems Research Laboratory  
Department of Computer Science  
Johns Hopkins University*

## Abstract

OpenCM is a configuration management system that supports inter-organizational collaboration, strong content integrity checks, and fine-grain access controls through the pervasive use of cryptographic naming and signing. Released as an alpha in June 2002, OpenCM is slowly displacing CVS among projects that have high integrity requirements for their repositories. The most recent alpha version is downloaded 45 times per day (on average) in spite of several flaws in the OpenCM's schema and implementation that create serious performance difficulties.

Since the core architecture of OpenCM is potentially suited to other archival applications, it seemed worthwhile to document our experiences from the first year, both good and bad. In particular, we review the original OpenCM schema, identify several flaws that are being repaired, and discuss the ways in which cryptographic integrity has influenced the evolution of the design. Similar design issues seem likely to arise in other cryptographically checked archival storage applications.

## 1 Introduction

OpenCM [SV02a, SV02b] is a cryptographically protected configuration management system that provides scalable, distributed, disconnected, access-controlled configuration management across multiple administrative domains. All of these features are enabled by the pervasive and consistent exploitation of cryptographic names and authentication. Originally prompted by the needs of the EROS secure operating system project [SSF99], OpenCM is slowly displacing CVS among projects that have high integrity requirements on their repositories.

The essential goals of the OpenCM project are to provide proper support for configurations, incorporate cryptographic authentication and access controls, retain the “feel” of CVS [Ber90] to ease transition, provide exceptionally strong repository integrity guarantees, and lay the groundwork for later introduction of replication support. Performance is not a primary design objective, though we hope to keep the user-perceived performance of OpenCM comparable to that of CVS.

Security and storage architects use the term “integrity” to mean slightly different things. A storage architect is concerned with recovering from failures of media or transmission. A security architect is additionally concerned with detecting modification in the face of active, knowledgeable efforts to compromise the content. Because OpenCM is used to manage trusted content (such as trusted operating systems and applications), one of our concerns is that a replicate server controlled by an adversary might be used as a vehicle to inject trojan code. A key design objective has been to provide “end to end” authentication of change sets even when the replicating

host is hostile [SV02a]. This prompted us to adopt a secure schema based on cryptographic (non-colliding, non-invertible) hashes rather than cyclic redundancy checks as our integrity checking mechanism.

One result is that an OpenCM branch can be compromised only at its originating repository. When truly sensitive development is underway, a suitable combination of host security measures and repository audit practices should be sufficient to detect and correct compromised branches. We do not suggest that such auditing practices are pleasant, nor are they necessary to run an OpenCM repository for most purposes. The point is that they are *possible*, and for life-critical or high assurance applications the ability to perform such audits is essential. We are not aware of any other source configuration management system with this capability.

Though the techniques have long been known, delta-based storage, cryptographic naming, and asynchronous wire protocols are not widely used in current applications. The importance of asynchrony in latency hiding is similarly well known, but it is difficult to use in practice because it does not follow conventional procedure call semantics. Practical experience with the *combination* of these techniques is limited. OpenCM has exceeded most of our goals during its first year of operation, but it has also suffered its share of design errors and object lessons, many of which are described here.

While we anticipated most of the causes of our mistakes, we failed in several cases to fully understand how various design features would interact. As a result, some of our expectations about how to solve these problems proved misleading. One year later, the basic archival object stor-



age model behind OpenCM appears to be validated, but we are now able to identify some issues (and solutions) that are likely to impact other applications built on top of this type of store. Most of the issues we have found can be viewed as factoring errors in our schema design. In each case, the need to plan for delta storage and cryptographic naming was the unforeseen source of these mistakes. Taken individually, none of these schema mistakes are particularly unique or surprising. Taken collectively, we believe they paint a somewhat unusual picture of the constraints encountered by a cryptographically checked store.

While there are significant benefits to the use of cryptographic names, there are also costs. Hashes are significantly larger than conventional integer identifiers and do not compress. They therefore interact with delta-based storage mechanisms, impose potentially large size penalties in network object transmission, and can create unforeseen linkages between client and repository. Cryptographic hashes provide strong defenses against hostile attempts to modify content. A corollary to this strength is the difficulty of *legitimate* modification – for example when schemas need to be updated. We will discuss each of these issues in detail.

Online delta compression strategies rely on identifying common substrings between two objects for the purpose of computing a derivation of one from the other. In addition to problems with the particular delta storage strategy we adopted (an “improved” version of Xdelta [Mac00]), there are some unfortunate interactions between delta compression and cryptographic names. We will discuss these and their implications for schema design.

Though asynchronous messaging is a well-known solution for avoiding network round trips and hiding network latency, OpenCM does not lend itself readily to this approach. Some of the issues are inherent in cryptographic naming, while others are inherent in the schema of OpenCM (and probably *any* configuration management system).

While each of the areas we discuss has been a source of inconvenience for OpenCM users, we emphasize that each creates problems of performance (storage size, computation, or wire latency) rather than problems of integrity. Over the last year, we have had occasion to uncleanly terminate our repository server on many occasions. During that time, we have observed only two storage-related errors in the OpenCM alpha releases. One was a serializer error, while the other was the result of an internal computation generating incorrect values in an object. We got lucky on the second error in that it didn’t actually get triggered in the field; repair might have proven very difficult *because* of the integrity properties of the store. We will discuss this at length and the provisions that we made in the schema to make repair of this form possible when nec-

essary.

The balance of this paper proceeds as follows. We first review the design of OpenCM, providing a basis against which to measure our experiences. We then identify those areas in which OpenCM has been an unqualified success – most notably in the area of integrity preservation. We then describe in sequence the issues that have arisen from cryptographic names, mistakes in the original schema, delta storage, and communication. We also describe the changes that are being made as a result of what we have learned.

## 2 Design of OpenCM

Before turning to a discussion of successes and mistakes, it is useful to provide a sense of how the OpenCM application is structured. The schema described here has evolved from that described in earlier OpenCM papers [SV02a, SV02b]. Earlier descriptions characterized the application as having a single unified schema, but noted our belief that the repository schema could be cleanly distinguished from the schema of the OpenCM application. The description that follows reflects that separation.

### 2.1 The RPC Layer

OpenCM is a client/server application. The client and server communicate via a wire protocol implementing an extended form of remote procedure call (RPC) [Nel81, BN84]. The low-level OpenCM RPC protocol extends conventional RPC by allowing the server to respond with an arbitrary sequence of optional “infograms” before sending a response to the original request. The final response may consist of either an exception or a response:

- request
- ← infogram\*
- ← reply-or-exception

The ability to transmit exceptions is a reflection of the design of the OpenCM runtime system. Though the application is written in C, we have extended the language using macros to provide an exception handling system. In support of this, OpenCM application is built using the Boehm-Weiser conservative garbage collector [BW88] in place of a conventional memory allocator.

**Lesson:** Given our past experience with the SGI VIEW debugger and other distributed applications, we went to some care to design infograms into the protocol from the beginning. The idea was to allow clients in a future protocol version to send a series of asynchronous “send” requests terminated by a normal RPC:



→ SendEntity name  
 → ...  
 → SendEntity name  
 → Flush  
 ← EntityGram (Infogram)  
 ← EntityGram (Infogram)  
 ← OK (Response)

Using an asynchronous protocol, trip delays can be reduced where the application's control flow is not inherently data dependent. In hindsight, OpenCM's control flow is much more data dependent than we realized, and this feature will have limited value. Fortunately, this functionality carries no performance penalty.

## 2.2 Core Repository Schema

The OpenCM application is constructed on top of a repository schema that is largely application-neutral. The repository layer handles versioning, permissions, authentication and (eventually) replication.

### 2.2.1 Mutable, Revisions

The OpenCM store may be imagined as a versioning file system whose content model is a graph of immutable connected objects. The primitive schema of the repository consists of *mutables*, *revisions* and *buffers* (Figure 1). The

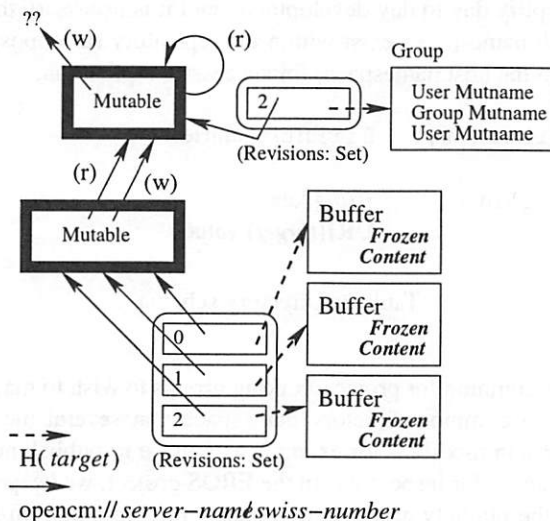


Figure 1: Core schema.

Buffer object serves as the univesal “envelope” for serialization of all content objects. The Mutable corresponds approximately to a “file” or “document” in a conventional file system. Each Mutable has a `readGroup` and `writeGroup` fields identifying the users and groups

that can perform the respective operations. Since the basic mutating operation on a Mutable is “revise,” members of the `writeGroup` are also allowed to read the object.

Revisions are made by first uploading a new content graph and then performing a `ReviseMutable` operation on some Mutable object to create a new Revision record. There is no direct equivalent to a file system “root” directory; the repository keeps a record of the set of authorized users. Each user has a corresponding Mutable that describes their authentication information and names their top-level directory object in the repository. The live state of the repository is defined by the set of transitively reachable objects beginning from the set of current users.

Figure 1 uses two different types of arrows to indicate two different types of names. Content objects are named by the cryptographic hash of their canonically serialized form (written as  $H(target)$ ). This hash can be recomputed by the repository without deserializing the object. Mutable objects are named by a URI of the form:

`opencm://server-name/swiss-number`

Where *server-name* is the cryptographic hash of the server's initial signature verification key, and *swiss-number* is a cryptographically strong random number chosen by the server at the time of object creation. The swiss-number has meaning only with respect to objects from the same server – there is no risk of collision across servers. As a shorthand, we will use the notation **URI(User)** to refer to a mutable whose content is an object of type User in the rest of this paper.

Revision records are named by appending their revision number to the mutable URI.

**Integrity and Security** The cryptographic hashes inherently provide an unforgeable check on their content. Since all pointers in the OpenCM content graph are cryptographic hashes, the top-level hash stored in the Revision record provides a transitive check on the entire content graph. Content-based naming also allows the server to store repeated objects once.

For mutables and revisions, integrity and security are provided by digital signatures. The mutable or revision is first serialized in canonical form and an RSA signature is computed using the server's signing key (written  $S(target)$ ). Both objects have a field reserved to contain this signature. The signature is computed with this field set to NULL. Provided that the OpenCM client has access to the server's signature verification key (the server's public key), compromises of mutable or revision content can be detected. Until a repository changes its signing key, the key's validity can be checked by computing the cryptographic hash of the server public key and comparing it to the server name. The client stores this association with an

IP address on initial connection in much the same way that OpenSSH does [Ylo96]. A technique for validating subsequent changes to the repository's signing key has been described elsewhere [SV02a].

The net result is that the signature checks on the initial mutable and revision records provide a complete end to end integrity check of the content graph associated with that mutable. All of this is handled transparently by the OpenCM client application.

**Lesson:** In principle, the repository layer has no need for specific knowledge of the application semantics. In practice, we did not attempt to separate these layers fully in the first implementation. Our thought was that we would achieve a better separation after building at least one real application. In hindsight, this was a good decision, because before building the OpenCM application we did not fully understand the layering interactions between the repository and the application.

**Lesson:** There is no forward linkage (pointer) in the schema from mutable objects to their revisions. In the earliest versions of the primitive schema, revision records contained a predecessor field, and a linked-list traversal was required in order to locate earlier revisions. Each step in the traversal required a round trip in the wire protocol, and it is frequently true that the client knows exactly which revision they require. The revision record was altered to facilitate faster retrieval of specific versions, and to permit selective replication of versions.

## 2.2.2 Authentication and Permissions

The repository is responsible for connect-time authentication of users. Our current authentication mechanism is built on SSL3/TLS [DA99]. The client presents the server with the public key of the connecting user. The server serializes this key as a PubKey object, computes  $H(\text{PubKey}(\text{user-public-key}))$ , and uses the result as an index into its user database. This index contains a set of  $(H(\text{PubKey}), \text{URI}(\text{User}))$  pairs, from which the identity of the per-user mutable for that user is obtained.

The per-user mutable has the same permissions structure as any other mutable. When adding a new user, the repository sets up their mutable to have self-modify access, and to be readable by the distinguished group Everyone. This allows users to alter their own read group. Note that the user's home directory is *not* initially readable by Everyone. At the time users are created, other users can see their existence, but not their work.

The User object is in turn a mapping from the user's public key to their "home directory." Every repository maintains a directory hierarchy for each user in which objects

can be bound with human-readable names.

User:     PubKey key  
           URI(Directory) directory

Group:    URI(User or Group) members[]

Table 1: Authentication-related schema

Groups are simply a set of URIs for other users or groups. Group membership is transitive: if a user is a member of group *A*, and group *A* is a member of group *B*, then the user is also a member of group *B*. In practice, this transitivity is not often used. It is based on an earlier permission system designed for the Xanadu global hypertext system [SMTH91], and is designed to allow a limited form of delegation for purposes of project administration.

## 2.2.3 Directories

Every user has a home directory that serves as the root of their accessible state. OpenCM directories provide a shared namespace mechanism that is controlled by the OpenCM permissions scheme rather than the host permissions scheme. Cryptographic monikers are not especially readable; directories serve a critical role as a human-readable name space. For collaborative groups that straddle many administrative organizations, this can greatly simplify day to day development, and it is necessary that such namespaces exist within the repository (as opposed to in the host namespace) for successful replication.

Directory:   DirEnt[\*] entries

DirEnt:       string key  
              URI(target) value

Table 2: Directory schema

It is common for project working groups to wish to maintain a common directory namespace that several members can modify – for example as a place to publish new branches for inspection. In the EROS project, we keep all of the publicly accessible branches in a common directory, and bind this directory into the directories of all of the project team members.

**Lesson:** Directory objects did not need to be part of the OpenCM core schema. They became part of the core repository to resolve a bootstrapping problem: when a User object is initially created, what content should its initial Revision object point to? In hindsight, the correct answer is "allow NULL to be a well-defined value for the

content pointer in a Revision record.” If this is done, the responsibility for creating, maintaining, and binding directories can be taken entirely out of the repository code. Once created, their handling is no different from any other mutable. The “tip off” is that only the user creation code manipulates Directory objects.

**Lesson:** Readers familiar with remote file system designs such as 9fs [PPT<sup>+</sup>92] or NFS [SGK<sup>+</sup>85] might be tempted to think that directory traversal should be performed on the server for reasons of protocol efficiency. OpenCM uses a distributed namespace in which the client may be able to resolve URIs that are not visible to the server. For this reason, directory traversal must be performed on the client.

### 2.3 Application Schema

OpenCM is a configuration management application layered on top of the OpenCM core repository schema. Its content schema consists of a very small number of object types (Figure 2).

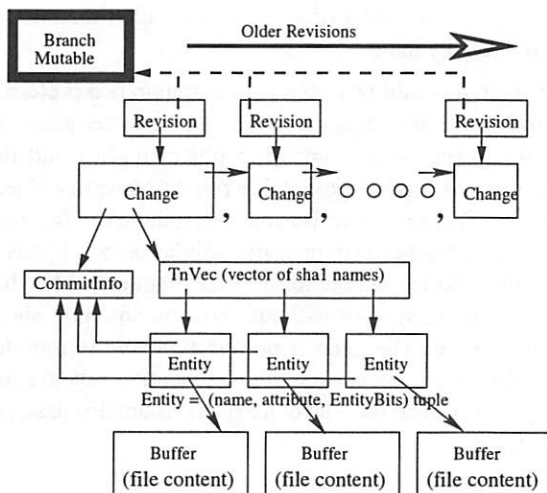


Figure 2: Application schema.

No explicit Branch object is needed in the OpenCM application, because the underlying repository keeps a history trail for every mutable object. Every revision results in a new Change, and each Change object records a new configuration for this line of development. Entity objects and Change objects both record predecessor pointers, but the former are omitted from the figure for clarity. The history of a branch is therefore encoded in two ways: through the predecessor hashes in the Entity and Change objects, and through the revision records of the per-branch mutables.

**Lesson:** Figure 2 shows the *current* object diagram for

OpenCM. In the original schema, the vector of Entity objects was inlined into the Change object. This was a source of tremendous performance loss, and is discussed in Section 5.2.

### 3 Successes

With the exception of performance, the initial implementation of OpenCM has met its overall goals extremely well. Using SSL3/TLS for authentication [DA99] makes it easy to allow “guest” users write access to our repository. Our hope that this would ease integration by replacing the patch application with merges has been validated. One of the OpenBSD team members has maintained the OpenBSD port in a branch that we periodically re-integrate. The ability to successfully hand this task off and smoothly re-integrate the results has greatly simplified maintenance. This is particularly important given that we do not run OpenBSD and are therefore unable to test these results. We have had similar assistance with ports for MacOS X and IRIX. We have likewise used the merge mechanism within the EROS project group, allowing each developer to play in their own branch before merging working code back into the tree.

Subjectively, our attempts to preserve the feel of CVS have been successful. The OpenCM command interface is not a clone of CVS. We preserved the most common commands in similar form, but regularized options across the board and favored a sensibly structured command set over compatibility. Users of CVS quickly find that they are comfortable with OpenCM. Subjectively, it is somewhat painful to revert to the less regular command structure of CVS. This result meets our goals on all fronts: we want it to be easy for developers to adopt OpenCM and we want them to notice when they are forced to revert to less functional tools. Several users report that they have simply switched their new projects over to OpenCM.

The integrity objectives for the OpenCM repository have also been achieved, and the results have exceeded our expectations. Because our naming and signing mechanism provides an end to end integrity check on the content of all objects, it is extremely difficult for data to be corrupted by the store or the network transport layer without detection.

We knew abstractly that cryptographic hashes provided a simple way to validate that an object is correctly retrieved. Only in actual use have we come to appreciate how useful this is. It is possible in OpenCM to garble the wire protocol, damage (by hand) objects in the store, or accidentally upgrade or modify objects whose content was not supposed to be modified. The validation ability inherent in the use of cryptographic hashes has caught all of these errors early in development. The principle sources of bad repository state in OpenCM have been a buggy serializa-



tion package (an escaping error in string encoding) and errors in the merge algorithm that caused objects to be incorrectly created (but stored with high integrity). In spite of crashes both deliberate and otherwise, we are not aware of any instance in which an OpenCM server or transport has undetectably corrupted the state of an object.

Our research group has been using OpenCM since before the day of its release. While there were a few early struggles as bugs emerged in the first few alphas, the tool has proven remarkably stable overall. Generally we are aware of flaws before they are reported by outside users. It is somewhat encouraging that a large percentage of the requests we receive are feature enhancement requests rather than bug reports. In many cases the desired function exists already, but is provided in a form that is not apparent to the requestor.

Perhaps the best indicator of success has been the reactions from other members of the open source community. We have been contacted informally by developers associated with nearly every major open source operating system to ask whether OpenCM could be incorporated into their release. For reasons that we are about to describe, our answer has uniformly been “yes, but doing so now would be premature.” The OpenCM alpha has intentionally been a long process so that we could determine what problems the design would encounter. Once the tool is widely dispersed, repairs to the kinds of problems we describe here will become quite difficult. Our goal in the long alpha cycle was to minimize the later pain to others that might be caused by early mistakes such as the ones reported here.

## 4 Issues with Cryptographic Names

OpenCM uses cryptographic names because they provide a universally non-colliding namespace that can be computed without global agreement or connectivity. The goal is to have an object naming system for frozen objects that can be replicated without collision even though the individual systems may have been disconnected at the time objects were created. By using a content-derived name, the replication engine process is able to efficiently determine by comparing object names which objects must be replicated.

While cryptographic hashing and signing provide exceptionally strong integrity, they also create some challenges.

### 4.1 Space Issues

Given the total number of objects that a given repository can be expected to store, a 64 bit (8 byte) object identifier would be more than large enough to name all of the objects in the repository. By comparison, a 27

byte cryptographic name (after base64 encoding) carries a large size penalty, and does not compress at all. Examination of the OpenCM application schema will reveal that the majority of bytes in the objects comprise these cryptographic names. As there is no hope of compressing these monikers, any space recovery must come from other mechanisms.

To achieve better space utilization, OpenCM uses delta-based storage. Each object is stored as a delta relative to some existing object, and is reconstructed on demand (Section 6). In order for delta-based space recovery to be effective, fields that are unlikely to change from one object version to the next should be stored adjacent to each other. This creates a maximal length identical byte sequence for the delta generation system to exploit for compaction. To date, we have approximated this by hand-ordering the fields in the object and preserving that order in the (de)serializer routines. In a larger scale application, the order of fields in the object is best determined by what makes sense to the programmer. In consequence, we suggest that an object definition language of some form should be applied and annotations should be used to indicate the preferred order of serialization rather than maintaining them by hand.

In abstract, it would be possible to maintain *two* preferred serializations simultaneously: the canonical serialization used for purposes of computing object hashes and the one used to maximize exploitable repetition across object versions. This proves to be counterproductive. In order for the server to perform integrity validation on objects it must either (a) know how to serialize them or (b) be able to recompute their hash without knowing anything about the object type. The latter is preferred, as the former destroys the separation between the application schema and the repository schema. The consequences are discussed in Section 7.3.

### 4.2 Name Resolution Issues

A previous paper [SV02a] on OpenCM identified the need for a secure name resolution mechanism to support server authentication. The resolution mechanism must provide a robust mapping from the server's name  $H(S(OrigVerifyKey_{server}))$  to its current IP address and signature verification key. This mapping is needed because the verification key and/or location of a repository may change over its lifespan. We have considered a number of designs for this secure name resolver, and remain partial to the one described in [SV02a].

To date, we have not found that the absence of this registry is a significant issue. In practice, users connect to repositories that are known to them *a priori*. The client records the server name and signature verification key of



that server on first connect. One of the delays in the implementation of distributed OpenCM is that this mechanism will break down completely when distribution occurs. Once server *A* is permitted to store a reference to an object on server *B*, it will become possible (indeed commonplace) for users to transparently dereference object names whose servers they have never contacted before.

A concern here that we did not adequately credit in our initial design is that the necessary scale of the name resolver is potentially comparable to that of DNS. Whoever runs the target name resolver is likely to be quickly overwhelmed. While client-side result caching is likely to be more effective in OpenCM than in DNS – each user contacts only a small number of servers and remembers them once contacted – we are reluctant to undertake the maintenance of such a large, availability critical data set. A question that arises is: how can we distribute the data set in such a way that the server containing the *reference* to the object is usually able to supply an up to date name resolution for the server that is actually *hosting* the object. That is, we would like on reflection to establish a two tier hierarchy in which OpenCM servers act as proxies for their clients in the name resolution process.

The scenario that really concerns us is as follows. Imagine that a new release of EROS is announced on Slashdot. Millions of people want to obtain the release (if we weren't optimists, we wouldn't have built a new configuration management system either). Most of these users have never seen EROS before, so their first step is to attempt secure name resolution. The problem is to prevent the name resolver from being overwhelmed – should this occur, then the run on EROS downloads would effectively preclude access to *other* repositories as well. DNS can be used for the IP-resolution portion of the problem, but does not provide a standardized means of distributing per-service keys. This problem awaits satisfactory resolution (sic).

### 4.3 Versioning Issues

For reasons discussed below (Section 5.2), certain types of schema changes cannot be made in a backwards-compatible way. Incompatible schema changes are rare, but they do not arise capriciously. Each has been introduced to solve a compelling problem with the OpenCM application. This has two unintended consequences:

1. OpenCM will tend to resist “forks” in the development process. As the body of useful repositories grows, there is considerable pressure to avoid competing rewrites of the object schema, and therefore to stick with a centrally coordinated OpenCM application.

2. Incompatible upgrades will tend to be “all at once.” If repository *A* replicates content from repository *B*, a schema upgrade on *A* will tend to force clients of *B* to upgrade their copies of the OpenCM software to understand the new schema.

The replication itself can proceed without upgrading the *B* repository server, and lines of development originating on *B* can still be accessed with older OpenCM clients. The current repository garbage collection strategy would need to be replaced with a conservative approach in order for “blind” replication (i.e. replication where the replicating server doesn't know the application schema associated with the content) to be practical. This is one of the reasons that we chose a relatively self-describing format for our encoding of hashes and mutable names.

## 5 Mistakes in the Schema

While cryptographic hashes guarantee integrity, they also prevent *intentional* changes to objects. If a field is added to an object, existing objects cannot be rewritten to incorporate the new field. Instead, object types must be versioned and the application must know how to forward-convert older objects into the current format. As long as new fields have a reasonable default value, this is straightforward, but object refactorings will generally require a rewrite of the repository to implement an incompatible schema change. In the history of OpenCM to date we have encountered both issues.

### 5.1 Backward Version Compatibility

The original `Entity` schema was missing a modification time field; we believed initially that storing the creation time of the entity (i.e. the checkin time) was sufficient. In practice this proved mistaken, because some commonly used build procedures rely on correct restoration of modification times on checkout. For example, it is common practice for tools using `autoconf` to include makefile dependencies designed to regenerate configuration files on demand. To eliminate `autoconf` version dependencies, these same projects ship the *output* of the `autoconf` script in their distributions. If modification times are preserved, `autoconf` will not be reinvoked.

Unfortunately, if modification times are *not* preserved, `autoconf` is invoked and versioning conflicts between the `configure.in` file and the locally installed copy of `autoconf` can arise. The issue is particularly acute with the `autoconf` package itself, which uses `autoconf` to perform its own configuration and includes an automatic regeneration target in its makefile.

We were able to compatibly add modification times to the

existing *Entity* schema by choosing the creation time of older *Entities* as the default modification time in the absence of better information. This resolved the problem with *autoconf*, but it left us with an unintended consequence: upward compatibility is insufficient; at least partial downward compatibility is also required.

The issue arises because *OpenCM* stores copies of the *Entity* objects in the client-side workspace file for later reference when performing merges. To avoid identity mismatches, the *Entity* written to the workspace must be byte-identical to the one originally obtained from the repository. If forward conversion occurs by inserting default values in new fields, these fields must *not* be rewritten when the object is reserialized. The original object serialization format included both an object type and a version number of that type, but our original in-memory object structure did not preserve the version number. Fortunately, we were able to modify the in-memory form to preserve the version number without altering the serialized form of our objects.

## 5.2 Refactoring Change

Before the work described in this paper, the *Change* object contained a vector of *Entity* objects:

```
Change:  Entity entities[*]  
        H(predecessor Change) preds[2]  
        H(CommitInfo) commitInfo
```

The earliest schema contained a vector of *Entity hashes*, but a very common operation in *OpenCM* is to request a *Change* object and immediately request all of its member *Entity* objects. Each frozen object request requires a round trip on the wire, which made this sequence prohibitively expensive. Having initially failed to adequately consider the role of asynchronous messaging requests in our protocol design, We inlined the *Entity* objects before the first alpha release of *OpenCM* in order to reduce the total number of round trips. This proves to have been a mistake for three reasons:

1. We later realized that by recording additional data in the *Entity* object we could frequently eliminate the *Change* object traversals altogether by walking the *Entity* predecessor chain. Inlining the *Entity* objects deprived us of the ability to exploit this.
2. The resulting *Change* objects are difficult to delta-encode. Where the hashes of the *Entity* objects can be sorted before writing to maximize exploitable repetition, the objects themselves have just enough differences to defeat delta compaction.
3. As the project grows, the *Change* objects become

measured in megabytes. As both merge and update algorithms need to traverse the history of *Change* objects, this results in a serious performance problem – even at broadband connection speeds.

We have therefore revised the *Change* objects to place even the vector of names in a separate object, modifying the *OpenCM* client to fetch individual *Entity* objects only as they are needed.

Unfortunately, this change requires a grand rewrite of the repository, because it is not a problem that can be straightforwardly handled by version-sensitive serialization logic. The deserializer could clearly replace the *Entity* vector by a vector of corresponding hashes, but subsequent attempts to fetch those objects would fail because they were never stored as individual objects in the repository.

If we had to do so, we could “repair” this problem by reading the existing *Change* objects, extracting their *Entity* members and rewriting those separately, but given that we need to do a rewrite of the repository anyway, we have decided to do an in-place conversion. This is better than carrying legacy support for the wrong schema in the post-alpha *OpenCM* client, and allows us to test the server rewriting process while the experience is still survivable.

The original decision to inline seemed plausible for several reasons:

- In the absence of a subsequent enhancement to the *Entity* schema, the application reference pattern suggested that this inlining decision was a good call.
- In the absence of protocol asynchrony, adding a round trip delay for each *Entity* fetch had been a source of performance issues.
- We had briefly tried a vectorizing fetch operation, but concluded (wrongly) that this imposed undesirable memory overheads and potential resource denial of service issues on the server. We failed to recognize early that asynchrony would enable request flow control, which would in turn resolve the problem of resource denial.
- We wished to avoid introducing into the wire protocol a request of the form “send all *Entity* objects referenced by this *Change* object,” because we wanted to avoid building application specific knowledge (in this case, of the *Change* object) into the wire protocol.

In actual fact, the real problem is none of the above. A properly designed asynchronous request, possibly combined with a length-limited vectorizing fetch operator,

would have addressed both the round trip problem and the server memory overhead problem. The underlying issue is that the `Entity` object is only four or five times as large as its cryptographic name, so the number of bytes needed to *request* the desired `Entity` objects is considerable. If fetching all of the `Entity` objects in a `Change` were indeed as common as it initially appeared, the wire overhead of the additional fetch requests would not be not justified.

**Lesson:** The lesson here is that object schema designs in a cryptographically named object store must take into account both the delta encoding effects and the actual usage patterns of the application. This was a case of premature optimization. It would have been relatively easy to inline the vector later. Inlining it early is going to cause us to rewrite all existing repositories.

## 6 Issues in the Store

OpenCM repositories can be based on flat files or an XDelta variant called SXD. Where XDelta uses backward-encoded deltas, the SXD implementation chose to use forward encoding. The idea behind forward deltas was to allow new object encodings to include references to fragments from all previous objects in the same delta container. We were forced in any case to re-implement the basic XDelta algorithm because the existing implementation had an I/O streams design incompatible with the one we used elsewhere in OpenCM and also because it was crafted in C++. More than 17 calendar years of experience with C++ leads us to conclude that code written in C++ is nearly impossible for ordinary programmers to maintain and presents runtime compatibility issues on some platforms.

**Lesson:** With the benefit of hindsight, this decision was simply foolish, and the lesson is that we should have read the literature. The analysis of alternatives performed by MacDonald in the design of XDelta [Mac99], as well as the measurements performed in connection with VDelta [HVT96], clearly indicate that reverse deltas are more efficient than forward deltas in nearly all cases. Our own rough experiments confirm that this decision alone added 40% to overall storage consumption. To add to our embarrassment, it is plain in hindsight that the issue of self-referencing copy operations in the delta encoding is completely orthogonal to the question of forward versus backward deltas.

Fortunately, the output of the delta encoding strategy is independent of the generation strategy. It has been possible for us to revise this decision incrementally without externally visible impact outside the repository.

## 7 Issues in Communication

In the course of implementing the cryptographic hash computation, the OpenCM runtime introduced the notion of typed I/O streams. These encapsulate an interface to the object I/O layer. Just above this sits the serialized data representation (SDR) library, which can read/write data in a number of encodings ranging from binary to a human-readable text encoding. There are three types of output streams: streams to files, streams to buffers, and streams that append their content to a pending hash computation but do not record the actual content.

Since OpenCM is a data motion intensive application, the stream implementation is performance-critical. Our original implementation was created in haste, and had approximately a factor of 20 in unnecessary overhead.

**Lesson:** Code in haste, repent forever.

### 7.1 Compression

Because of the nature of the OpenCM request stream, it is likely that there is re-referenceable content from one request to the next. In implementing the OpenCM wire protocol, we made the mistake of compressing on a per-request basis rather than simply using a compressed stream. In hindsight this was a mistake. The connection, rather than the messages, should be compressed as a stream. We suspect that the loss of compression opportunities resulting from this decision is substantial, but have no numbers to demonstrate this.

### 7.2 Inefficiencies in Hashing

Given the existing SDR library, it became natural to specify the definitive method for computing the hash of an OpenCM frozen object as follows:

1. Serialize the object to a `Buffer` stream using the SDR layer.
2. Compute the cryptographic hash on the resulting byte string.

Further, it is natural to perform inbound object I/O by reading the object into a `Buffer`, because the buffer implementation takes care to avoid unnecessary memory re-allocations.

This specification for hash computation has a desirable side effect: given a buffer containing the raw bytes of an object, the naively computed cryptographic hash of the buffer content is the cryptographic hash of the object.

Ironically, we failed to notice this for several months. Instead of specifying the `GetEntity` wire invocation to



return a `Buffer` object, we specified it to return a pointer to an object of the expected type. Because this object is what the server needs to return, we actually reload the byte representation of the object from its `SXD` container into a `Buffer`, deserialize this `Buffer` back into an object, and then *reserialize* the object to a hashing stream to check its integrity. A new RPC call returning a buffer delta has been introduced, and the old call will shortly be retired.

A second (and arguably more significant) benefit to manipulating objects purely at the `Buffer` level is improved ability to stream data motion through the server. There is no reason for the server to verify the object hash before returning the object to the client. Since the client must check the hash in any case, a more efficient design would simply pass the `Buffer` through without examination. The current implementation does so. Note that this facilitates streaming of large objects. Since the server *should* handle objects larger than the available address space (but currently does not), the ability to stream objects in this way is a potentially critical change in the design.

### 7.3 Client/Server Version Lockup

The mistake in the specification of the `GetEntity` wire invocation has a second consequence: it forces the schema version numbers of client and server to remain identical. Because OpenCM clients in practice talk to multiple servers and servers talk to multiple clients, this effectively demands that all installations of OpenCM upgrade when a new version of OpenCM is released.

After some initial settling, the wire protocol itself has not proven to be a source of versioning issues. All integers traversing the wire as part of messages (distinct from object content) use only strings, and unsigned integer values. We encode unsigned values as length-independent strings, avoiding possible compatibility issues resulting from integer length. The OpenCM client and server negotiate a wire protocol version at the start of each connection. This has allowed us to extend existing wire operations compatibly by doing version-sensitive encoding and decoding of the operations.

The source of the problem lies in the present need to deserialize and reserialize each object on its way through the server. The current `ServerRequest` and `ServerReply` messages directly serialize the object instead of simply passing along the containing `Buffer` (the envelope) as an opaque object. In order to perform the deserialization and reserialization, the server must know the type of the object being transmitted, which exposes the server to schema changes in the application.

In practice, the only frozen objects that the server *needs* to know about are the `Mutable`, `Revision`, `Buffer`,

`User`, and `Group` objects. For reasons explained in Section 2.2.3, the server currently knows about `Directory` and `Dirent` objects as well, but use of these objects is restricted to the “create user” function of the repository. The server is responsible for creating and initially populating the new user’s home directory. After creation, the server performs no further interpretation of `Directory` or `Dirent` objects.

By changing the wire protocol specification to return `Buffers` rather than the objects they contain, a hierarchical layering of schema dependencies emerges:

1. If the underlying wire schema is upgraded in a way that cannot be made backwards compatible, then everybody must upgrade. To date we have successfully avoided this in most cases.
2. If the server schema is upgraded, then all clients connecting to that server must upgrade. We have not made significant changes to the server schema since it was originally deployed.
3. If the application schema is upgraded, older servers remain able to act as “carriers” for newer application-layer payload. There remain possible compatibility issues between two copies of the OpenCM client; older clients may be unable to read objects placed on the server by newer clients.

### 7.4 Failure of Asynchrony

In the original OpenCM design, we included infograms in the wire protocol specification but did not use them. One of the authors had built several heavily asynchronous applications in the past, and we were very much aware of the benefit of asynchrony.

We were also aware of the fact that *useful* asynchrony is limited by the application semantics. If the nature of the algorithms in the application create significant data-driven control dependencies, blocking is dictated by the application-level semantics and cannot be eliminated at the wire level. In OpenCM, it has proven that the majority of requests made by the client are dependent on the results of the immediately preceding request. So far, there are only two exceptions:

- When building a merge plan, the client retrieves the top two objects to be merged (these can be either `Change` objects or `Entity` objects, depending on the merge phase) and performs a breadth-first ancestor walk in order to find the nearest common ancestor of the two entities. The search expansion step could be performed using asynchronous requests.



- When performing a checkout or update, the client retrieves the current `Change` object and immediately retrieves all of the associated `Entity` objects in that `Change`. This can be vectorized, as can the later fetch of the `Buffer` object containing the file content for those files that are needed.

The combined impact of these two changes is potentially significant, but it is not immediately clear how they will interact with client-side caching. In merge operations this optimization is essential, but in other operations the dominant cost in using OpenCM is the SSL cryptography setup cost.

## 8 Measurements

The changes described here have impacted both the repository size and the operational speed of OpenCM. The size improvements come primarily from the introduction of the SXD2 repository format. The speed improvements come primarily from the stream logic changes and the `Change` schema modifications.

All of the numbers that follow are shown for the *local* (direct file access) repositories. Given the reduction in `Change` size, the new schema should show network performance improvements as well, but the major improvement in network performance will not be measurable until we introduce the wire delta transmission and asynchronous requests described in Section 7.

**Space** OpenCM now supports three storage formats: flat files, gzipped files, and SXD2 archive files. The flatfile format is used primarily for testing purposes. A typical store has a bimodal distribution of object sizes. Table 3 shows object size distributions for a test repository holding the complete checkin history of OpenCM itself and for the our main repository (which contains EROS, OpenCM, and various internal projects). The only objects in the store larger than 1 block are the file content and the `TnVec` objects of Figure 2 (the EXT3 file system uses 1 kilobyte blocks). The `TnVec` objects are not compressible at all, and the majority of objects are already one block or smaller, so the achieved compression of 37.5% is actually quite good.

The schema change alone reduced the size of our main repository significantly. While surprising, this number is correct. The EROS project has more files in each configuration than the OpenCM project. In OpenCM, the reduction in stored metadata was offset by the changes to content. In the EROS project, the reduction in stored metadata dominates the total repository size. The EROS project dominates the main repository.

The SXD2 format combines multiple frozen objects into

	objects ≤ 1 block	objects > 1 block	disk use 1k blocks
Test flat	5,121	2,968 (18,944)	94,932
Test gz	5,230	2,859 (6,992)	59,332
Main flat	115,802	55,265 (22,907)	1,889,560
Main gz	132,007	39,060 (13,980)	1,207,416

Table 3: Object size distributions and total disk use under the new schema. Parenthesized numbers show average object size among objects larger than 1 kilobyte.

delta-based archive files, thereby reducing internal fragmentation within the file system. Disk utilization for SXD2 is shown in Table 4. The SXD2 storage format remains highly compressible. Initial estimates collected while gathering these numbers suggest that another 42% reduction in overall storage is possible.

Schema:	old	new	new	new
Store:	gz	gz	SXD2	mut+rev
Test	48,076	59,332	24,736	2,104
Main	1,745,632	1,207,416	874,832	32,572

Table 4: Repository storage in disk blocks under various schemas and storage schemes. The “mut+rev” column shows the subset of disk blocks occupied by mutables and revisions, which are stored as gzipped files in all formats shown.

A key factor in reducing total SXD2 storage size is to store indexing structures using a DBM (or similar) file-based format to gain efficient space utilization. If (e.g.) symbolic links are used, total SXD2 storage is *higher* than storage using gzip, because each symbolic link requires a full disk block. Unfortunately, the GDBM library performs updates in place on the existing file, and this introduces risk of index corruption if operations are interrupted. Fortunately, the SXD2 archiver *never* performs in-place modifications, and the DBM index can be reconstructed from the archive files. We are considering storing revision records and mutables in a similar fashion (using SXD2 and DBM). While we expect no delta compression, the improvement in file system utilization from reduced internal fragmentation is probably worthwhile.

**Performance** To measure performance, we checked out revisions of OpenCM that were 100 revisions old, and then timed three versions of OpenCM performing the update. The first version of OpenCM uses the old schema. The second incorporates only the serialization library improvements. The third includes serialization improvements and uses the new schema. The principle cost of

the update is walking the Change hierarchy to locate the “nearest common ancestor” in order to compute the necessary update. The results are shown in Table 5.

Version	User	Supervisor
Old	5.11s	0.39s
New Serializer	1.12s	0.33s
New Schema	0.97s	0.41s

Table 5: Update performance

The “new schema” improvements come from two sources: the size reduction of Change objects and an optimization in the merge code that eliminates unchanged Entity objects from further consideration without fetching them. This optimization significantly reduces the number of objects that must be fetched in the update process, and consequently will reduce the total number of network round trips.

## 9 Related Work

There is a great deal of related prior work on configuration management in general. We focus here only on architecturally related systems or systems that are being used by open source projects. Interested readers may wish to examine the more detailed treatment in the original OpenCM paper [SV02b] or various other surveys on this subject.

### 9.1 CM Systems

**RCS and SCCS** provide file versioning and branching for individual files [Tic85, Roc75]. Both provide locking mechanisms and a limited form of access control on locks (compromisable by modifying the file). Neither provides either configuration management or substantive archival access control features. Further, each ties the client name of the object to its content, making them an unsuitable substrate for configuration management.

**CVS** is a concurrent versioning system built on top of RCS. It is the current workhorse of the open source community, but provides neither configurations nor integrity checks. A curious aspect of CVS is that it has been adapted for use as a software distribution vehicle via CV-Sup [Pol96]. This directly motivated our attention to replication in OpenCM.

**Subversion** is a successor to CVS currently under development by Tigris.org [CS02]. Unlike CVS, Subversion provides first-class support for configurations. Like CVS, Subversion does not directly support replication. Subversion’s access control model is based on usernames,

and is therefore unlikely to scale gracefully across multi-organizational projects without centralized administration.

**NUCM** uses an information architecture that is superficially similar to that of OpenCM [dHHW96]. NUCM “atoms” correspond roughly to OpenCM frozen objects, but atoms cannot reference other objects within the NUCM store. NUCM collections play a similar role to OpenCM mutables, but the analogy is not exact: all NUCM collections are mutable objects. The NUCM information architecture includes a notion of “attributes” that can be associated with atoms or collections. These attributes can be modified independent of their associated object, which effectively renders every object in the repository mutable. NUCM does not provide significant support for archival access controls or replication.

**WebDAV** The “Web Documents and Versioning” [WG99] initiative is intended to provide integrated document versioning to the web. It is one of the interfaces used by Subversion, which allows strong web integration. WebDAV provides branching, versioning, and integration of multiple versions of a single file. When the OpenCM project started, WebDAV provided no mechanism for managing configurations, though several proposals were being evaluated. Given the current function of OpenCM, OpenCM could be used as an implementation vehicle for WebDAV.

**BitKeeper** incorporates a fairly elegant design for repository replication and delta compression. To our knowledge, it does not incorporate adequate (i.e. cryptographic) provenance controls for high-assurance development. Further, it does not address the trusted path problem introduced by the presence of untrusted intermediaries in the software distribution chain. In contrast to current implementations of all of the previously mentioned technologies, BitKeeper’s license does not facilitate community involvement in improving the tool, and has been a source of controversy in the open source community.

### 9.2 Other

Various object repositories, most notably Objectivity and ObjectStore, would be suitable as supporting systems for the OpenCM repository design. This is especially true in cases where an originating repository is to be run as a distributed, single-image repository federation. Neither directly provides an access control mechanism similar to OpenCM.

Both Microsoft’s “Globally Unique Identifiers” and Lotus Notes object identifiers are generated using strong random number generators. Miller *et al.*’s capability-secure scripting language *E* [MMF00] uses strong random num-

bers as the basis for secure object capabilities.

The Xanadu project was probably the first system to make a strong distinction between mutable and frozen objects (they referred to them respectively as “works” and “editions”) and leverage this distinction as a basis for replication [SMTH91]. In hindsight, the information architecture of OpenCM draws much more heavily from Xanadu ideas than was initially apparent. The OpenCM access control design is closely derived from the Xanadu Clubs architecture [SMTH91], originally conceived by Mark Miller.

OpenCM’s use of cryptographic names was most directly influenced by Waterken, Inc’s *Droplets* system [Clo98]. Related naming schemes are used in Lotus Notes and in the GUID generation scheme of DCE.

## 10 Acknowledgments

Mark Miller first introduced us to cryptographic hashes, and assisted us in brainstorming about their use as a distributed naming system. The directory system in OpenCM is indirectly based on the “pet names” idea that he invented while at Electric Communities, Inc. Josh MacDonald took time for a lengthy discussion of OpenCM and PRCS in which the strengths and weaknesses of both were exposed. It can fairly be said that this conversation provided the last conceptual validation of the idea and prompted us to go ahead with the project. Various discussions on the Subversion mailing list pointed out issues we might otherwise have failed to consider. Paul Karger first pointed out to us the requirement for traceability in building certifiable software systems, and the fact that given then-current tools this requirement was incompatible with open-source development practices.

At the risk of forgetting others who may deserves mention, Todd Fries and Jeroen van Gelderen stand out as particularly active among the outside contributors to OpenCM. Each has made significant contributions to OpenCM’s improvement, and has assisted us in hunting down some obscure, irritating, and difficult to track bugs. Interaction on the `opencm-dev` mailing list has been a significant source of improvements and ideas.

## 11 Conclusion

Delta-based storage, cryptographic naming, and asynchronous wire protocols are not widely used in general purpose applications. Though the importance of asynchrony in latency hiding is well known, asynchronous wire protocols do not interoperate easily with conventional procedure call semantics, and are therefore difficult to use in practice. Practical experience with the *combination* of these techniques is limited. In this paper, we have

tried to report on our experiences building a highly robust application combining these techniques.

OpenCM has suffered its share of design errors in the first year. In several cases the decisions we made in the design interacted with the use of cryptographic naming and delta compression in unforeseen ways, and these have taken time to sort out. These interactions proved particularly tricky in the layering of the OpenCM schema and in their performance consequences. At this point, we have identified most of the problems, and are in the process of deploying their solutions.

On the whole, the underlying design approach of the OpenCM repository seems sound. Cryptographic naming is a significant but manageable contributor to storage, communications, and runtime overhead in schemas involving small objects. In repositories storing larger objects such as documents, this issue would be significantly less important and the robustness and modification controls supplied by OpenCM would be significant.

OpenCM is built on a small set of simple ideas that are pervasively applied. While there are many interdependencies in the design, the only complex algorithm or technique is the merge algorithm. The key insights are that successful distribution and configuration management can be built on only two primitive concepts – naming and identity – and that cryptographic hashes provide an elegant means to unify these concepts.

Taken overall, OpenCM has met or exceeded nearly all of our expectations. The pain of correcting the errors described here has more to do with the sheer scale of the application than with the complexity of the repairs. We have lived with the tool extensively for a long time now, and none of us would go back to other open source tools given a choice.

The core OpenCM system, including command line client, local flatfile repository, gzipped file repository, SxD2 repository, and remote SSL support, consists of 25,794 lines of code – a 27% growth over the last year. Much of this growth has occurred to complete existing function and clarify the merge algorithm. In contrast, the corresponding CVS core is over 62,000 lines (both sets of numbers omit the diff/merge library). In spite of this simplicity, OpenCM works reliably, efficiently, and effectively. It also provides greater functionality and performance than its predecessor. One of the significant surprises in this effort has been the degree to which a straightforward, naïve implementation has proven to be reasonably efficient.

OpenCM is available for download from the EROS project web site (<http://www.eros-os.org>) or the OpenCM site (<http://www.opencm.org>). A conversion tool for existing CVS repositories is part of the distribution.



## References

- [Ber90] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, CA, 1990. USENIX Association.
- [BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, pages 39–59, 1984.
- [BW88] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, pages 807–820, 1988.
- [Clo98] Tyler Close. Droplets, 1998.
- [CS02] Ben Collins-Sussman. The subversion project: Building a better cvs. *The Linux Journal*, February 2002.
- [DA99] T. Dierks and C. Allen. The TLS protocol version 1.0, January 1999. Internet RFC 2246.
- [dHHW96] A. Van der Hoek, D. Heimbigner, and A. Wolf. A generic peer-to-peer repository for distributed configuration management. In *Proc. 18th International Conference on Software Engineering*, Berlin, Germany, March 1996.
- [HVT96] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. An empirical study of delta algorithms. In Ian Sommerville, editor, *Software configuration management: ICSE 96 SCM-6 Workshop*, pages 49–66. Springer, 1996.
- [Mac99] J. MacDonald. Versioned file archiving, compression, and distribution, 1999.
- [Mac00] J. MacDonald. File system support for delta compression, 2000.
- [MMF00] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Proc. Financial Cryptography 2000*, Anguila, BWI, 2000. Springer-Verlag.
- [Nel81] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, May 1981.
- [Pol96] J. Polstra. Program source for cvsup, 1996.
- [PPT+92] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. In *Proceedings of the 5th ACM SIGOPS Workshop*, Mont Saint-Michel, 1992.
- [Roc75] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.
- [SGK+85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Proc. 1985 USENIX Annual Technical Conference*, pages 119–130, Portland, Ore., June 1985.
- [SMTH91] Jonathan S. Shapiro, Mark Miller, Dean Tribble, and Chris Hibbert. *The Xanadu Developer's Guide*. Palo Alto, CA, USA, 1991.
- [SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, December 1999. ACM.
- [SV02a] Jonathan S. Shapiro and John Vanderburgh. Access and integrity control in a public-access, high-assurance configuration management system. In *Proc. 11th USENIX Security Symposium*, San Francisco, CA, August 2002. USENIX Association.
- [SV02b] Jonathan S. Shapiro and John Vanderburgh. CPCMS: A configuration management system based on cryptographic names. In *Proc. FREENIX Track of the 2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002. USENIX Association.
- [Tic85] Walter F. Tichy. RCS: A system for version control. *Software – Practice and Experience*, 15(7):637–654, 1985.
- [WG99] E. James Whitehead, Jr. and Yaron Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the web. In *Proc. of the Sixth European Conf. on Computer Supported Cooperative Work (ECSCW'99)*, Copenhagen, Denmark, September 12–16, 1999, pages 291–310, 1999.
- [Ylo96] Tatu Ylonen. SSH — secure login connections over the Internet. In *Proc. 6th USENIX Security Symposium*, pages 37–42, 1996.



# Free Software and High-Power Rocketry: The Portland State Aerospace Society

James Perkins     Andrew Greenberg  
Jamey Sharp     David Cassard     Bart Massey  
Portland State Aerospace Society  
Computer Science Department  
Portland State University  
Portland, OR USA 97207-0751

james@psas.pdx.edu     andrew@psas.pdx.edu  
jamey@cs.pdx.edu     dcassard@psas.pdx.edu     bart@cs.pdx.edu  
<http://psas.pdx.edu>

## Abstract

The Portland State Aerospace Society (PSAS) is a small, low-budget amateur aerospace group. PSAS is currently developing medium-sized sub-orbital rockets with the eventual goal of inserting nanosatellites (satellites that weigh less than 10 kg) into orbit. Because achieving orbit requires a navigation system able to guide the rocket along an orbital trajectory, PSAS is pioneering an open source and open hardware avionics system that is capable of active guidance.

In this paper, we describe how free software and open hardware have dramatically changed the capabilities of amateur aerospace groups like PSAS. We show how we have applied existing and custom free software to the avionics and ground support systems of a sub-orbital sounding rocket, and discuss what further work must be done.

We conclude that the sophistication and complexity achieved by current amateur avionics projects—which are beginning to challenge the distinction between amateur and professional—would not be possible without the use of free software.

## 1 Overview

This paper details the role of free software and open hardware in our group, the Portland State Aerospace Society (PSAS). First we introduce the field of rocketry, and how PSAS is making contributions to the field. This is followed by a history of the group's rocket development and an overview of our current project. We then cover the details of our system, including requirements, flight software, ground software, and the project management and collaboration tools. Finally, we discuss our future work, and draw conclusions about the applicability of free software to projects like ours.

## 2 Introduction to Rocketry

Rocketry has a rich history of significant contributions made by amateur groups. Indeed, most national space programs can trace their roots back to small, active groups of amateurs working on rocketry during the early part of the 20th century [Win83].

Amateur rocket motor classifications are categorized by their total impulse, which is total force multiplied by burn time. "A" motors have up to 2.5 Newton-seconds (Ns) of total impulse, and each successive letter of the alphabet doubles the impulse range of the previous letter [Tri].

Today, rocketry includes a wide spectrum of participants and can generally be divided into four categories.

*Model rocketry* involves the smallest and most common types of rockets, with motors made of pressed black powder that are smaller than 320 Ns ("H"), and bodies built out of balsa wood and cardboard or phenolic (resin) tubes. Most model rockets weigh less than a kilogram and are recovered with a small parachute or streamer. Since model rockets generally stay below 460 m (1,500 ft), they avoid most government regulation in the US.

*Hobby rocketry* begins from the upper limits of model rocketry. Motors delivering up to about 10,240 Ns ("M") of thrust are common. Motors are typically made of composite fuel (ammonium perchlorate in a HTPB plastic binder), and are sometimes clustered or staged to achieve greater power. Some high-powered hobby rockets can reach more than 6 km (20,000 ft), so launches in the US are government regulated and require launch waivers. Most hobby rockets use commercially available, single board avionics systems that sense the best time to eject parachutes [alt], although some custom-built avionics packages have been much more sophisticated.

*Amateur rocketry* usually implies a certain level of innovation and customization, such as custom motors, custom avionics and metal airframes. This innovation may be inspired by a lack of suitable commercial solutions or a desire to “do it yourself”. The current altitude record for an amateur group is 85 km (280,000 ft) with a 327,680 Ns (“R”) motor [rrs].

*Professional rocketry* is the most familiar category. It includes organizations such as NASA and the European Space Agency (ESA), along with their private contractors such as Lockheed Martin, Boeing, and Orbital Sciences. Over time, the line between amateur and professional rocketry has blurred. Some industry observers discriminate by financial gains, technical expertise, or sheer altitude achieved, but there is no common standard. Indeed, as technologies such as computational power and integrated sensors become cheaper and more available, the capabilities of amateur groups have begun to catch up to some professional projects.

### 3 The Portland State Aerospace Society

The Portland State Aerospace Society (PSAS) was founded in 1997 by two students at Portland State University (PSU) in Portland, Oregon to provide an aerospace-based, systems-level educational design project. PSAS has launched one hobby and two high-powered amateur rockets. The group has grown to include more than three dozen people, including high school, undergraduate and graduate students, as well as engineers from local industry. Current PSAS projects are focused on taking small, manageable steps towards the distant vision of inserting nanosatellites (satellites that weigh less than 10 kg) into orbit.

Is it possible for amateur rocketry groups to achieve orbit? None have, to date. In most countries, the regulatory hurdles are at least as much of a challenge for an amateur group as the substantial technological and financial issues. The US Commercial Space Launch Act of 1984 requires any rocket with over 890,000 Ns (“S”) of impulse lasting for 15 seconds or more to meet stringent safety requirements set by the FAA, NASA, and Commercial Space Transportation Board (CSTB) [csl, sls]. These requirements include an extensive safety analysis, which can take years and cost hundreds of thousands of dollars. Furthermore, vehicles which actually achieve orbit must comply with international “space law” as laid out in various international agreements [un].

While fulfilling the goal of achieving orbit may be beyond the ability of our small group, the enabling technologies needed to get there by an amateur group are now readily obtainable: inexpensive computational power, sophisticated sensors, high-power actuators, and the availability of robust, open source software for engineering and logistical support.

### 3.1 Toward Amateur Active Guidance

To achieve orbit at minimum cost, a rocket must follow an “orbital trajectory” that minimizes both the aerodynamic drag in the lower atmosphere and the time to get to orbit [Wer92]. To follow such a trajectory, the rocket must be able to measure its current trajectory, compare against the planned trajectory, and actively correct for errors. This ability to follow a trajectory is called “active guidance”, and the authors know of no amateur group that has yet achieved this. To bridge this gap in amateur rocketry technology, PSAS has chosen to work on open source and open hardware high-powered amateur rockets that are capable of active guidance.

There are many meanings of active guidance. Here we use active guidance in the classic rocket sense: a guidance computer on-board the rocket measures the rocket’s current position, heading and course. The guidance computer then activates a steering mechanism in order to guide the rocket along a predetermined path.

To determine the rocket’s position, attitude (orientation) and trajectory (flight path), a rocket’s flight computer uses data from one or more sensors:

- GPS receivers provide absolute position at a slow ( $\sim 1$  Hz) rate.
- Inertial Measurement Units (IMUs) provide relative linear and rotational acceleration, velocity and position using accelerometers and gyroscopes at a faster rate ( $\sim 1$  kHz).
- Magnetometers provide attitude (orientation) by comparing the local magnetic field to a 3D map of the Earth’s magnetic field.
- Pressure sensors allow altitude to be computed from atmospheric pressure models.
- Optical sensors, such as star and horizon trackers, may be used to compute attitude.

Each of these sensors have different signal and noise characteristics. Kalman filtering [CC99] is a signal processing algorithm for combining noisy sensor data that provides guarantees on the optimality of its estimate. This estimate can then be used as an approximation of the rocket’s true position, attitude and trajectory.

To steer the rocket, some mechanism must apply a force to the rocket during flight. Steering mechanisms vary widely:

- Small fins are common, but are ineffective above about 25 km (82,000 ft).
- Reaction Control Systems use small rocket motors to adjust the heading of the rocket, but are usually large, heavy systems that are ineffective in the lower atmosphere.
- Thrust Vector Control changes the angle of the main motor’s thrust, for example by gimballing the main motor nozzle, by independently throttling

multiple clustered motors, by putting small movable vanes in the path of the exhaust, or by injecting extra oxidizer into the edge of a fuel-rich exhaust stream [SB00].

To “close the loop”, estimates of position, attitude and trajectory are used to calculate steering commands. The most difficult part of active guidance may be getting the signal processing in the control loop correct. Many famous rocket failures, such as the first launch of Ariane 5, were due to simple errors in design or implementation of the navigation system algorithms [ins].

## 3.2 Related Work

Many amateur groups exist today, but none that the authors know of are currently working on active guidance. Guided amateur designs do exist, but all either track the sun or are used to simply stabilize the rocket in flight. None can actually be used to follow a planned trajectory. An excellent example of a guided, but not actively guided, rocket is the MARS Rocket Society’s gyroscope-controlled gimballed-nozzle rocket [mar].

## 3.3 Launch Vehicles

Because we strongly emphasize safety, reliability and new functionality for every launch, PSAS has launched only four times over its six year history. In a field with failure rates as high as 30% at some events, we have not yet lost a vehicle. Further, each new launch has demonstrated new airframe or avionics functionality which justified the time and expense of performing a launch.

### 3.3.1 Launch Vehicle No. 0

The first PSAS project began with four volunteers and a simple hobby rocket dubbed “Launch Vehicle No. 0” (LV0). The team modified a commercial kit of cardboard tubes and balsa wood with fiberglass and epoxy resin and added a simple avionics system.

While the airframe took only a weekend to complete, the avionics system took two people a few months of building and testing. The avionics system had just a few components: a 1 MHz 8 bit RISC microcontroller; a micro-electro-mechanical (MEMs) accelerometer; a 426 MHz 1 W amateur television (ATV) transmitter; a monochrome video camera; and a commercial logging altimeter. The interrupt-driven microcontroller firmware was written in assembly language. Accelerometer samples were downlinked using the ATV audio channel at 300 bps. Telemetry data was then logged on a 386 DOS laptop.

LV0 was launched on June 7, 1998 (Table 1). Not surprisingly, things went wrong. The airframe was all but unstable and a short circuit in the wiring harness erased system memory. However, the received images and data proved the soundness of the overall system concept.

### 3.3.2 Launch Vehicle No. 1

LV0’s success attracted more volunteers: about a dozen people designed and built the next generation rocket, Launch Vehicle No. 1 (LV1). The airframe was made of carbon fiber over a PVC and aluminum core, and was built by one person over 3 months, with little engineering analysis.

The LV1 avionics system was arguably the most advanced amateur rocket avionics package in the world in 1999. Its subsystems included:

- A custom flight computer board with an 8 bit 33 MHz RISC microcontroller, 1 MB of non-volatile SRAM for data logging, pyrotechnic ignition circuitry, and interface circuitry to other subsystems including temperature and pressure sensors.
- An inertial measurement unit (IMU): X, Y, and Z axis linear accelerometers and yaw, pitch, and roll rate gyroscopes. These sensors produced a “six degree of freedom” measurement which was numerically integrated to calculate the rocket’s 3D position and velocity.
- A commercial 12-channel GPS receiver.
- A color video camera and microphone that transmitted over a 426 MHz ATV transmitter.
- A 913 MHz 1 W transmitter that sent 19.2 kbps telemetry data encoded by the flight computer.
- A 146.43 MHz (2 m) amateur radio receiver with a DTMF decoder. Decoded tones were sent to an independent microcontroller that could fire the recovery system or send commands to the flight computer in an emergency.

The flight computer’s assembly-language, interrupt-driven executive sampled all sensors, logged data, and transmitted low-pass filtered telemetry. The firmware also used the GPS, IMU and pressure sensors to determine the rocket’s altitude and thus when to deploy the recovery system.

LV1 required extensive ground support due to its size and complexity. A surplus pneumatic lifter was modified into a launch tower with a 6 m (20 ft) launch rail. Launch control software automatically performed the countdown and launched the rocket via a 903 MHz wireless link to a small microcontroller-run relay board.

Another Linux-based ground computer captured, logged and displayed live telemetry data with a custom GTK-based application. This helped the range safety officer decide if the flight was proceeding as planned: if not, the emergency radio could be used to deploy the recovery system (parachutes) even if the flight computer failed.

LV1 was launched on April 11, 1999 and again (as LV1b) on October 7, 2000 (Table 1). Half of the \$2,000



Table 1: Comparison of PSAS Launch Vehicles

Name	Size	Dia.	Weight	Ns	Altitude	Cost	Design	CPU	OS
LV0	1.8 m	10.0 cm	5.4 kg	J	0.3 km	\$500	4 mo.	PIC16C	Interrupt-driven
LV1	3.4 m	11.0 cm	19.5 kg	M	3.6 km	\$2000	18 mo.	PIC17C	Interrupt framework
LV2	4.0 m	13.6 cm	46.0 kg	P	23.0 km	\$10000	>21 mo.	5x86	Linux / RTLinux

development cost came from an IEEE/AT&T "Student Enterprise" grant.

LV1's complexity led the dozen-plus volunteers to divide into airframe, avionics and logistics teams. Each team had their own mailing list, web page and FTP folders. However, formal design methods were mostly ignored and the web site was updated infrequently. Designs were implemented without review, which caused schedule slips and two "scrubbed" launches.

### 3.3.3 Launch Vehicle No. 2

The next generation launch vehicle needed the flexibility, modularity and extensibility that LV1 lacked. A few of the lessons learned include:

- Real-time Operating Systems (RTOS's) hand-built in assembly language are not a good idea for a complex, multi-volunteer effort. An off-the-shelf RTOS with better networking and free, easily learned development tools is much more appropriate.
- The navigation software needs serious computational power: a CPU with hardware floating point support is highly desirable.
- Subsystems in the avionics system must be easily added, removed or swapped.
- The airframe itself must be flexible and expandable to handle unforeseen requirements.

The design of LV2 took more than a year of careful coordination between the avionics and airframe teams. The design process was much more formal than had previously been tried. A white paper was written on the various design options for the avionics system. The airframe team used finite-element analysis to predict the performance of different airframe structures. To fund the project, PSAS applied for and won the 2000 Oregon Space Grant, a \$10,000 NASA-sponsored small grant program.

The LV2 airframe uses a flexible and modular design to facilitate swapping out entire subsystems such as the avionics, recovery or propulsion (motor) unit. We have also separated the avionics system from the payload module, enabling LV2 to fly other academic or amateur rocketry payloads. The resulting airframe is made of cylindrical aluminum modules covered by a fiberglass aeroshell. Simulations predict a maximum altitude of 23 km (75,000 ft) (Table 1).

Based on a computational complexity analysis of the navigation software, we decided to use a flight computer

with a floating point unit (FPU), better support for multitasking, and a modern development toolchain. However, a single-processor avionics system was unappealing because of the high-rate, I/O-intensive tasks many of the sensors required: controlling a high speed analog-to-digital converter and running a closed loop motor controller, for example.

A multi-node common bus solves many of these problems by enabling a larger, more powerful central flight computer to communicate with many smaller microcontrollers. This allows a "smart sensor" and "smart actuator" approach that frees up the central processor to perform higher-level calculations and supervisory tasks.

We chose the Controller Area Network (CAN) bus as our intra-rocket multi-node bus. The CAN bus is an automotive bus developed by Robert Bosch, GmbH which is quickly gaining acceptance in both the industrial and aerospace markets. CAN is a multi-master, losslessly-arbitrated serial bus that can be run up to 1 Mbps. The CAN bus includes packet-level checksums and tolerance of node errors (including logic that forces a node off of the bus if it is causing errors). Perhaps the most interesting aspect of CAN is its message-based identification of packets: instead of node addresses, the CAN bus identifies and prioritizes the messages based on an 11 bit identifier. Each message, such as GPS location, or IMU inertial data, is broadcast on the bus with a unique message ID [can].

For the central flight computer we have selected the PC104 form-factor. This allows us to use standard off-the-shelf parts instead of taking the time and effort to make our own. The flight computer consists of:

- A Jumptec MOPS520 PC104+ board, a 133 MHz 5x86 processor (the AMD SC520) with 64 MB of SDRAM, typical PC ports, and a CAN interface.
- A carrier board for a 128 MB CompactFlash hard disk.
- A PCMCIA carrier board for a Lucent Orinoco 802.11b card.

Considerable difficulty was encountered in developing the wireless telemetry system for LV1. We thus wanted a commercially available long-range high-speed bi-directional wireless telemetry system for LV2: such a system could be used for telemetry as well as by the rocket, launch tower and ground computers. After some research, we gravitated toward the Amateur Radio Relay League (ARRL) 802.11b standard. ARRL 802.11b



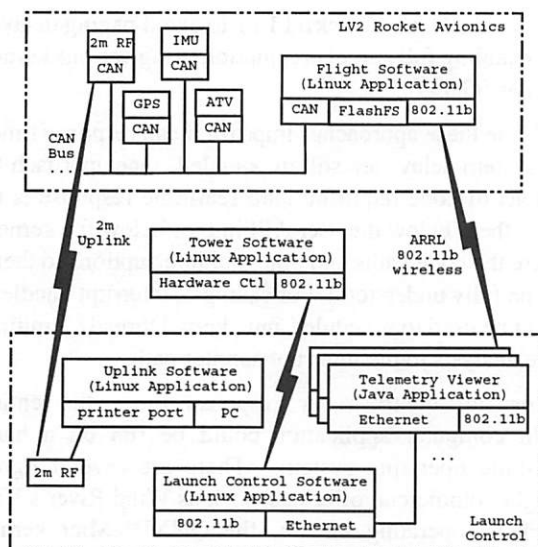


Figure 1: LV2 System Software

is the IEEE 802.11b 2.4 GHz spread spectrum standard, but operated under the FCC amateur radio regulations (FCC Part 97) instead of the low-power, unlicensed regulations of the 2.4 GHz ISM band (FCC Part 15). Running under ARRL 802.11b allows us to use up to 100 W of radiated power. Our current system uses a standard PC-Card (Lucent Orinoco), a 1 W bidirectional power amplifier, and high gain (+12 dB) helical ground antennas [arr].

The smaller sensor/actuator CAN nodes use the Microchip PIC18F458, an 8 bit, 40 MHz RISC flash-memory microcontroller with a built-in CAN protocol unit. There are currently five CAN nodes:

- The LV1 IMU.
- A SigTech Navigation MG5001 OEM GPS receiver.
- A power interface node to control system power and track battery charge.
- A recovery node—a battery-backed up CAN node with high voltage pyrotechnic firing circuitry. Like LV1, it has a 2 m amateur radio receiver which decodes DTMF tone sequences sent as emergency commands.
- An ATV node, consisting of a color video camera, ATV transmitter, power amplifier and an overlay board that displays textual vehicle status information along with NTSC flight video.

The CAN nodes have proven to be useful general-purpose building blocks: new nodes are frequently prototyped atop a generic “misc CAN node” design.

## 4 Software Overview

The software system is divided into three major areas:

- **Flight Software**
  - *Flight Computer Software* flies in the rocket, and runs on a single-board computer running Debian GNU/Linux (Section 5.1).
  - *Avionics Firmware* flies in the rocket, and runs on multiple independent microcontroller nodes (Sections 5.2–5.3).
- **Ground Software**
  - *Launch Control Software* is used by the flight control officer to sequence the rocket launch. *Launch Tower Software* manages launch tower electronics (Section 6.1).
  - *Telemetry Display Software* is used by the flight control officer and spectators to view the rocket’s status. (Section 6.2).
  - *Uplink Software* is operated by the range safety officer at launch control and sends emergency commands through the 2 m uplink (Section 6.3).
  - *Collaboration Software* runs on the (Portland State University) hosted PSAS server. The PSAS server is the locus for a variety of services including mailing lists, collaborative web pages and CVS repositories (Section 7).

## 4.1 Functional Requirements

There are a number of mandatory requirements for a successful rocket launch. In order of priority:

1. *Safety*. The risk of death, injury, and property damage must be minimized
2. *Reliable Recovery*. The airframe must be safely recovered. The principal software constraint is to deploy the recovery parachute only at apogee, when vertical airspeed is at a minimum.
3. *Flight Data Recovery*. The purpose of every flight is to collect new data: data recovery is thus essential.
4. *Telemetry Downlink*. Real time sensor data must be easily monitored by the launch controller to decide whether or not to override the flight computer via the emergency uplink.
5. *Radio Dropout Tolerance*. Radio links are notoriously unreliable. The software must be able to tolerate link failures.
6. *Recovery Assistance*. Position information must be available to the recovery teams tracking the rocket after the parachutes have opened. High winds or malfunction can carry the rocket kilometers away during ascent or descent.
7. *Power Management*. Intelligent power management is crucial to reliable performance. Power use also constrains peak altitude. Lithium batteries

store energy at a density of  $\sim 300$  WHr/kg. Thus, during a standard flight profile each watt used on the rocket adds 30 g (1.1 oz) of battery weight. Simulations predict that the altitude-to-weight ratio is  $\sim 335$  m/kg. Thus every watt used on the rocket means a reduction of 10 m (33 ft) of altitude.

## 4.2 Logistical Requirements

The volunteer nature of our group imposes some special project management requirements:

1. *Minimize Coordination Overhead.* In our project, collaboration is entirely ad-hoc. Project synchronization and access control mechanisms must reflect this.
2. *Minimize Training.* Well-known development methods are necessary to enable immediate contributions by our new volunteers.
3. *Limit Costs.* Funding is scarce: we must take advantage of any available cost reductions.

## 4.3 Choosing a Common Platform

Our requirements for a low-cost, reliable and flexible platform discourage the use of non-free, proprietary and unfixable software and suggest using well-documented, open-standards-based free software. GNU/Linux not only runs on the limited hardware we have available, but also enables the use of thousands of free software applications available for UNIX environments.

We have standardized on the Debian GNU/Linux distribution for all of our development, collaboration and rocket systems in order to reduce the amount of time spent maintaining and installing software [deb]. Thanks to Debian's Advanced Package Tool (apt), our systems generally require little maintenance effort while remaining relatively secure. This enables us to focus on our development efforts instead of struggling with platform and tool configuration. It also reduces the time necessary to configure development environments for our new volunteers.

## 4.4 From Soft to Hard Real-time

The term "real-time" refers to applications that require guaranteed bounds on the time between the occurrence of an event and the software's response. Soft real-time requirements allow for occasional missed deadlines under high load, but hard real-time applications must meet all deadlines under all conditions [Lab99].

Linux is designed to optimize throughput, not to guarantee hard real-time response. Processing may be delayed by mutexes, interrupt locks, high interrupt load, paging, and other causes. The scheduling latency of a Linux task may be improved using the `sched_setscheduler()` system call to identify it as a "real-time" task. Other tricks include us-

ing `mlock()` or `mlockall()` to avoid paging delays, and enabling full kernel preemption using various kernel patches [Gal95].

While these approaches improve mean response time, the system delays are still unbounded. One approach to regions of code requiring hard real-time response is to move them below the user API into or below the kernel, where the code paths and potential interruptions to them can be fully understood. For example, interrupt handlers might be used to schedule Linux kernel threads, limiting code analysis to the interrupt handler path.

For full control over response time, the entire flight computer application could be run on a hard real-time operating system. There are several light-weight commercial offerings, such as Wind River's Vx-Works<sup>TM</sup> operating system, the QNX<sup>TM</sup> Microkernel and Jean Labrosse's MicroC/OS-II. However, commercial RTOS's generally require specialized programming knowledge, often support a narrower array of devices than free software, and can be expensive.

Fortunately, the best of both worlds may be found in real-time operating systems that run the entire Linux kernel and all user processes in the lowest-priority thread. One such offering is FSMLabs Inc.'s RTLinux [fsm]. The hard real-time elements of our software can be implemented using the real-time kernel primitives: everything else runs as Linux user processes.

The PSAS flight computer application has a variety of requirements. Some components have no real-time requirement, while others have soft or hard real-time requirements. For the next launch, our application's real-time requirements are soft: launch and apogee must be detected and handled within a second, and data must be logged and transmitted to the ground as soon as possible. However, the introduction of navigation algorithms on future launches will require us to move to hard real-time, and we plan to begin using RTLinux with our Fall 2003 launch.

## 5 Flight Software

The LV2 flight system uses three significantly different kinds of processors (Table 2). This necessitates three significantly different software architectures: processes running in a 32 bit pre-emptive multitasking RTOS; threads running in a light-weight POSIX-compatible RTOS; and tasks running in a custom interrupt-driven framework. In this section we review these software architectures. Note that we distinguish firmware from software: firmware consists of small, hardware-oriented programs stored in nonvolatile memory and considered read-only by the local processor.

Table 2: Processors used in the LV2 Avionics System							
System Name	Arch.	Bits	Speed	RAM	Flash	MMU?	OS
Flight Computer	AMD 5x86	32	133 MHz	64,000 KB	128,000 KB	Y	Linux
GPS Receiver	ARM7TDMI	32	40 MHz	128 KB	1,000 KB	N	eCos
CAN Nodes	PIC18	8	40 MHz	2 KB	32 KB	N	PicCore

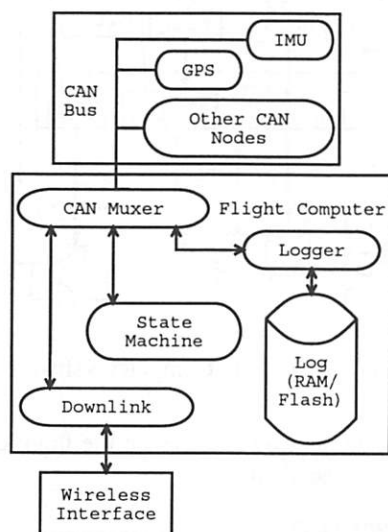


Figure 2: Muxer Flight Computer Software Design

## 5.1 Flight Computer Software

A key element of the application software running on the 5x86-based flight computer (see Section 3.3.3) is its message-passing architecture. This software is currently written in C, and runs in a Debian GNU/Linux environment with a Linux 2.4 kernel. Messages are passed across the wireless link, across the CAN bus and to the nonvolatile data log (Figure 1).

Every component of the PSAS avionics system has different response speed and latency characteristics. In order to avoid holding up the whole application, each device is monitored and controlled by one or more asynchronous execution tasks. Coordination of these tasks is accomplished by passing messages between them, initially by using a message server task. We have been through three flight computer software designs with different message-handling schemes: “Muxer”, “Renegade” and “FCFIFO”. After introducing these designs, we will discuss some other flight-computer software elements of interest: the CAN Bus driver and our network configuration.

### 5.1.1 Muxer

The first design was implemented in C using POSIX threads, TCP/IP sockets, and device I/O processes. The processes passed messages by connecting to a server process called *Muxer*, which broadcasted each message

received to every other task.

Muxer’s server thread began by initializing a shared buffer. It then listened for TCP/IP connections. The central data structure was a variable-sized shared queue of messages. When a client connection was accepted, the server thread spawned client queue reader and writer threads. The new client’s queue reader thread slept, waiting for new messages to arrive in the queue. The client’s queue writer waited for an incoming message and enqueued it. Queue access was serialized by a mutex, and semaphores were used to wake sleeping queue readers when new messages were waiting for delivery. The last client to read a message from the queue deallocated that queue element.

The Muxer design included a client library to ease creating a TCP/IP connection to the Muxer service. Clients were mostly other Linux processes on-board the flight computer or (via the wireless link) on the ground (Figure 2). A shell script first started the Muxer server process, then each of the client processes. The client processes would open and initialize their I/O device. They would then use the Muxer client library to open a connection to the Muxer server thread.

There were some difficulties with this design:

- *Excessive wake-ups.* The star message topology with N clients resulted in each new message waking up N threads, which wrote N messages to the network stack, which woke up N processes to read the data.
- *Message synchronization.* Messages were of variable size, making mid-reception synchronization to a message stream a challenge.
- *Debugging.* Debugging POSIX threads was difficult. Threads would sometimes damage the environment of their neighbors.
- *Opacity.* The queue structure, its mutex and use was not easily understood by the team.
- *Wrong protocol.* TCP/IP was a poor choice for wireless communications, incurring multi-megabyte queuing and retries in the network layer. Ultimately a downlink process was added to bridge the messages into a UDP protocol for the wireless downlink.

A CAN bus reader was implemented and Muxer was demonstrated to work. Nonetheless, when changes were introduced that broke Muxer, no one was able to find the time to sort out why. Team members eventually real-

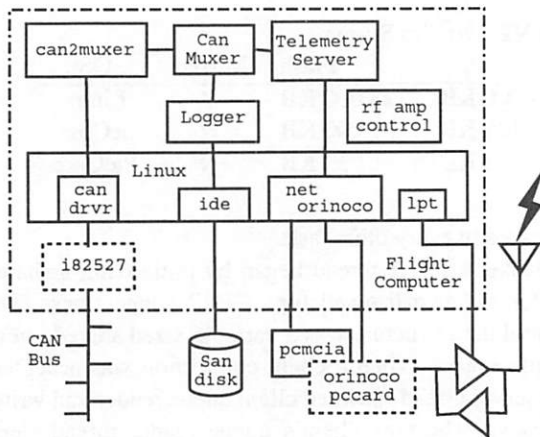


Figure 3: Renegade Flight Computer Software Design

ized that solving the flight computer design problem by starting from the message handling structure and then moving out from there was not working.

### 5.1.2 Renegade

One team member began working on an independent effort, and titled it the “Renegade” design. Renegade continued the theme of a central Muxer task dispatching each message to all asynchronous I/O-handling clients. However, the architecture was implemented as a UNIX process with UNIX pipes to child I/O processes (Figure 3).

The top-down, application-oriented and simplified design was an improvement. Most importantly, needed rocket functionality was quickly completed:

- CAN bus messages were read and distributed successfully to all I/O devices.
- Intertask communication became independent of networking.
- Wireless dropouts were handled efficiently rather than causing backups.
- Fixed-sized messages eliminated synchronization issues.
- Debugging with GDB and UNIX signals was trivial.
- The Linux kernel design could be relied on for optimized intertask performance.

Renegade did have shortcomings.

- The lack of message filtering and the broadcast architecture still led to excessive wake-ups.
- The inherited pipe structure made it difficult to restart I/O processes from the shell.
- UNIX pipes were difficult to inject data into for testing and debugging.
- No throughput tests were ever attempted.

After some consideration of Renegade’s shortcomings,

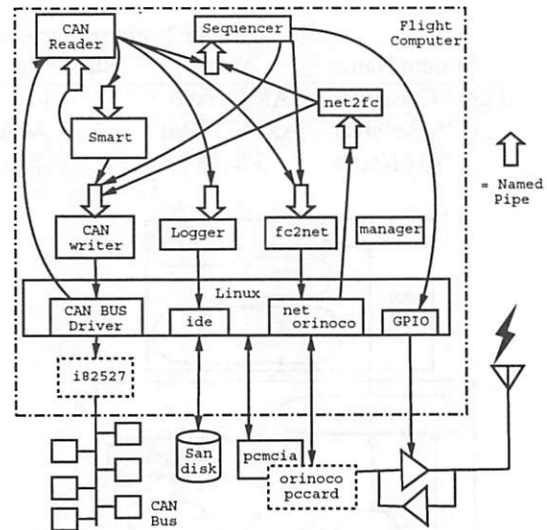


Figure 4: FCFIFO Flight Computer Software Design

we once again decided to redesign the flight computer software to correct them.

### 5.1.3 FCFIFO

The third and completed design is called FCFIFO. It combines the independent processes from the Muxer design with the conventional UNIX interprocess communications of Renegade. In addition, it eliminates the central multiplexer (Figure 4). On startup, each process opens a uniquely named pipe to read messages from, then opens the named pipes of any processes it needs to communicate with. By eliminating the central multiplexer, task switch and copying overhead are minimized.

A simple Linux user-mode process-oriented application will satisfy the June 2003 soft real-time requirements. While named pipes do not guarantee a response time, they do aid in maximizing throughput. They also allow processes to run asynchronously, because they can buffer a few messages if the reading process falls behind. As components transition to RTLinux hard real-time tasks, this architecture will be ideal: named pipes are the preferred means of communication between Linux and RTLinux tasks.

Where it is advantageous for activities to occur independently, the application implements these via separate processes.

- *Logger* takes messages on its named pipe and stores them in the nonvolatile log file.
- *CAN Reader* opens the CAN bus device and blocks until it reads a message either from the CAN bus or from its named pipe. It passes messages to the *Logger* and networking for storage, sensor information to *Smart*, and a few other events to *Sequencer*.
- *CAN Writer* opens the CAN bus device for writ-



ing and waits for incoming CAN messages on its named pipe. It then sends those messages out onto the CAN bus.

- *Fc2Net* waits on its named pipe for a message, then broadcasts the message to the wireless network using our UDP application protocol.
- *Net2Fc* initializes a UDP server port, then waits for a message. If a command comes from the ground via the wireless network, Net2FC passes the command on to other relevant processes.
- *Smart* evaluates raw sensor data and converts it into a more useful form. (We discovered that this is a difficult component to name. After much thought, we decided it is the “Smart Module Assembling Real-time Tasks”.) For example, GPS and IMU data are processed to obtain position, velocity, and acceleration information. Smart also detects events such as rocket launch, apogee, and touchdown.
- *Sequencer* decides when the system should transition from one rocket flight state to another, and generates the commands needed to set up the CAN nodes for the new state.
- *Manager* is analogous to the UNIX `init` process. It starts the whole set of named pipe children, restarts children that die unexpectedly, and manages a graceful application shutdown.

### 5.1.4 Linux CAN Bus Driver

CAN communication is performed through a GNU GPL 82527 CAN chip driver for Linux, written by Arnaud Westenberg and adapted to our the flight computer’s 82527 interface [lcb]. The driver allows applications to communicate with the CAN bus through the `/dev/can0` device node.

### 5.1.5 Network Configuration

During development, the flight computer communicates with development machines using its built-in Ethernet or local IEEE 802.11b wireless. In flight configuration it uses the ARRL 802.11b hardware described in Section 3.3.3. Standard Linux PCMCIA and Orinoco drivers configure and integrate the network hardware. However, a simple custom driver is needed to switch the system between IEEE and ARRL 802.11b modes: the 2.4 GHz bidirectional power amplifier can be turned on and off via two general purpose I/O pins on the flight computer.

The wireless network card is configured to use channel zero (2.400 - 2.450 GHz, which is within the amateur 13 cm band), IBSS ad-hoc networking mode, a fixed IP address from a private network address range, and the maximum power output available from the card. Enabling IBSS ad-hoc networking prevents the card from participating in auto-configuration via a management

device such as a wireless access point. The telemetry viewing computers at launch control are the only other device similarly configured, to limit performance-robbing traffic collisions.

All network transmissions between the rocket and ground control use UDP broadcast packets. This provides a low-overhead transmission path with no packet retries. TCP was briefly considered for rocket communications, but TCP requires a reasonably error-free communications medium. Given the extreme 23 km distance of the rocket, antenna geometry effects, attenuation due to the motor’s exhaust plume, and tracking error of the ground antennas, it cannot be assumed that the wireless link can support TCP communications.

The telemetry and control protocol is symmetric, time-stamped, and is logged on each end. This allows us to measure communications loss through post-flight data analysis. Periodic general status messages from the rocket allow the ground software to resynchronize key parameters rapidly even when communication is intermittent. A data payload checksum helps validate the quality of the received data, although invalid data frames are also logged to aid in link quality analysis.

## 5.2 CAN Node Firmware: ARM7TDMI

The Global Position System (GPS) receivers on-board LV2 are commercially available boards that use the Zarlink GP4020 GPS correlator chip [gpsb], a custom ASIC with a 12-channel correlator and an ARM7TDMI microcontroller core. The receivers come with closed-source proprietary firmware which runs the correlators, processes the satellite data, and transmits position and velocity information over a standard asynchronous serial bus.

Unfortunately, the navigation algorithms in commercially available GPS receivers fail to cope with the high dynamics of LV2 and will lose satellite lock when the vehicle exceeds 515 m/s, 18 km in altitude, or 8 g’s of acceleration. Because LV2 will exceed these limits and a locked GPS receiver is a critical part of our navigation system, we have started a separate project: developing free firmware for GPS receivers which use the GP4020 chip. Dubbed GPL-GPS [gpsa], the firmware is based on Clifford Kelley’s “OpenSource GPS” project [kel] and will allow the receivers to stay locked during highly dynamic flights. The open firmware will allow us to implement our own sophisticated processing techniques, including integrating the GPS and IMU into a GPS-aided inertial navigation system [FB99]. The firmware may also allow us to:

- Implement local differential GPS corrections via a similar GPS receiver on the ground in a known, static position. Enabling a local differential base station can yield positioning accuracies in the 1 m

Table 3: Embedded Operating Systems Comparison

Name	RT	POSIX	Free	Small	Effort
RTLinux	Y	Y	Y	N	1x
$\mu$ cLinux	N	Y	Y	N	2x
MicroC/OS-II	Y	N	N	Y	4x
$\mu$ ITRON	Y	N	Y	Y	4x
eCos	Y	Y	Y	Y	4x
Custom	Y	N	Y	Y	10x

range.

- Use multiple GPS receivers to determine attitude (orientation) by comparing the position of their antennas.
- Keep GPS lock on the satellites by integrating data from the inertial measurement unit to aid the correlator tracking loops.

We chose to use an existing RTOS in order to speed up firmware development, and because a multi-threading abstraction will simplify the GPS firmware design. GPL-GPS requires an RTOS with extremely low latency, POSIX compliance to facilitate porting, a reasonable free or open-source license, ability to run in 1 MB of flash memory and 128 KB of RAM, and a reasonable effort level (estimated as a time multiplier) for us to implement with our hardware. Based on our research (Table 3), we selected Red Hat's eCos (embedded Configurable Operating System) [Mas03] as the best fit to our criteria. We are currently working on the alpha release of the GPL-GPS project, which includes a port of eCos and Kelley's OpenSource GPS to the GP4020. We hope to have initial results by September 2003.

### 5.3 CAN node Firmware: PIC18

The Microchip PIC18F458 8 bit RISC microcontrollers [pic] used on the CAN nodes (see Section 3.3.3) have 1.5 KB of RAM, 32 KB of flash memory, and a long list of peripherals. The latter include an 8-channel 10 bit analog to digital converter, various serial interfaces including CAN, timers and an in-circuit debugging port. The PIC18 is used as a small and simple interface between sensors and the CAN bus, implementing a rudimentary "intelligent sensor and actuator" network. Because of the extremely tight memory constraints and the simple functionality of the firmware, we have written a custom C-language interrupt-driven framework called "PicCore".

Unfortunately, there are few free software applications for PIC18 development. For example, there is no GCC cross-compiler for the PIC18 family. The GDB debugger does not support Microchip's "ICD2" in-circuit debugging tool. Our current development environment consists of Microchip's no-cost Windows-based integrated development environment (MPLAB

IDE) [pic], graciously donated copies of HI-TECH Software's PICC-18 C compiler [C18], and Microchip's ICD2 debugger/programmer. Our failure to get MPLAB to run under emulators such as WINE unfortunately require us to have Windows-based development environments for the PIC18 developers. Developing our own PIC18 development tools would exceed the resources of our project, so for now we have resigned ourselves to operate with two development environments. We are currently searching for equivalent hardware that has better free software development support.

## 6 Ground Systems

The software and hardware components on the ground (as shown in Figure 1) have several objectives:

- Maintain the safety of all participants and bystanders.
- Initiate the launch sequence (with means of emergency abort).
- Receive and record telemetry data from the vehicle during flight,
- Receive and record video broadcast from the vehicle during flight,
- Initiate manual recovery procedures if the flight software appears to be failing.

Due to our small budget, our volunteer team has always relied on the kindness of strangers for available computing hardware. As a result, we need to be able to run our software on as many hardware and software configurations as possible. We have chosen Java as our ground systems language because Java byte-code enables us to run our code on any sufficiently powerful platform. Java's automatic memory management and simple GUI framework make ideal tools for user interfaces and data visualization. Java is well known: many of the developers in our group are familiar with it.

### 6.1 Launch Control Software

The launch control software is a Java application which steps through an automated launch sequence. The sequence of events is coordinated with the rocket and launch tower via ARRL 802.11b wireless links: the launch tower may be several kilometers away from launch control for safety reasons. Safety systems include manual interlocks and a rocket-controlled interlock triggered on flight computer diagnostic information. These interlocks prevent accidental launches in the event of software or hardware failure.

The launch tower computer is the same as the rocket's flight computer: a PC104 stack with 5x86 processor, 802.11b card, power supply and nonvolatile flash memory. The controller even has a CAN bus: a CAN-based relay board controls launch tower hardware such

as strobe warning lights, sirens, and the rocket motor ignition relay.

## 6.2 Telemetry View Software

When the rocket is at its peak altitude of 23 km (70,000 ft), it is almost impossible to visually ascertain what is happening. The "Rocketview" telemetry display software is used by flight personnel and interested bystanders to observe the rocket's status. The flight controller uses telemetry information to decide whether the flight sequence is proceeding according to plan: if not, the range safety officer can manually deploy the recovery system using the emergency 2 m radio uplink.

The Rocketview application displays data received from the rocket via ARRL 802.11b wireless link. The display includes 6 strip charts (X, Y, and Z position as well as roll, pitch and yaw attitude), several text fields, and a free-form console log for miscellaneous messages from the flight computer. The Rocketview application was modeled after a GTK-based C program used for LV1. Because Rocketview is written in Java, it can take advantage of Java's graphics capabilities and of such open source components as the JFreeChart charting tool [jfr].

## 6.3 Uplink Software

The uplink system is an emergency backup communication link. The link is manually activated by the flight controller in case the flight computer fails to deploy the recovery system. The uplink computer runs a Java application which communicates via serial port with a Yaesu 50 W 2 m amateur transceiver. Upon user command, the application keys the Yaesu transmitter and generates a series of DTMF tones (the familiar touch-tone telephone tones) which are received by the recovery CAN node on the rocket. Besides emergency commands, the uplink software can also generate a small number of CAN bus commands for diagnosis purposes.

## 6.4 Video Capture System

Amateur TV signals received from the color camera on the rocket are recorded on tape and displayed live on a monitor. The flight personnel and bystanders can thus observe the flight from the rocket's point of view. The rocket uses a video overlay board to display position and flight computer status information. Because the ATV transmitter is higher power and lower bandwidth than the ARRL 802.11b transmitter, this display functions as a backup telemetry downlink.

## 7 Collaboration Software

Before the LV2 project, PSU's Electrical and Computer Engineering department hosted a web site and Majordomo list server [maj] for PSAS activities. The site

was primarily used to promote recruitment and showcase progress. No centralized version control was used and only one or two people updated the web site. Nearly all engineering documentation was passed back and forth between individuals using private email and removable computer media. This collaboration model was ill-suited to the size and complexity of the LV2 project: it was too difficult for team members to share their efforts quickly and effectively.

To resolve these problems, PSU agreed to host a donated Pentium-class computer. By co-locating a PSAS-owned computer, we gained flexibility in service provisioning and the flexibility to set up and experiment with collaboration software. Four of the team members with significant system administration experience have administrative privileges. Anyone on the team can get OpenSSH [ssh] shell accounts for purposes including software development and document preparation. PSU creates a backup of this system nightly using Amanda [ama].

Mailman [mai], with its web interface, makes maintaining the public announcement, team coordination, and site administration mailing lists easy. In particular, mail list subscribers handle their own subscription and mailing list preferences. Although less than ideal, CVS [cvs] has made it possible to share and back up work on software and other development documents, even across different operating systems.

The PSAS web site (<http://psas.pdx.edu>) still serves a promotional purpose. However, it now also serves as an ongoing project development notebook, with meeting minutes, task lists, specifications, team work areas, back-of-the-envelope calculations and diagrams online. This is accomplished using TWiki [twi]. TWiki is a Perl CGI-based Wiki Wiki Web implementation by Peter Thoeny and others, comprising a web-based collaboration platform designed for corporate and academic intranets. Using TWiki, any team member can add or alter site content using a web browser. TWiki automatically generates hypertext links, allows for attaching multimedia content, supports website search, can be configured to automatically notify subscribers of changes and can remind users of upcoming or overdue action items.

Begging for installation of tools from University staff, or working through one or two authorized web secretaries using static HTML, would severely constrain our entire project. Free software collaboration tools make it possible for team members to contribute and collaborate at their own pace, which dramatically improves our team's productivity.



## 8 PSAS Contributions

To date, amateur groups building custom avionics have developed private software of little use to other groups. In contrast, the PSAS software design is modular, based on inexpensive and open hardware, licensed under the GNU General Public License, and available for download from our web site. Any interested team is welcome to use and build on our work.

We strongly encourage other amateur groups to use our TWiki site, mailing lists and CVS tools to collaborate with us on our projects: advancing the state of the art through wider community collaboration.

A few of our free software and open hardware projects available for use are:

- *Gerbertiler*. A Perl script which tiles together printed circuit board layout files. Tiling together many small boards into one large board significantly reduces the one-time fee associated with having a single board produced.
- *Misc. CAN node*. A 4 x 7 cm board for prototyping PIC18F458/CAN node applications. Complete schematics, board layout files and working code are available.
- *PicCore*. An interrupt-driven framework, peripheral APIs and network drivers for the PIC18F4xx family of microcontrollers, written using HI-TECH Software's PICC-18 C compiler.
- *GPL-GPS*. As discussed in Section 5.2, GPL-GPS is a project to create free firmware for any Zarlink GP4020-based GPS receiver board.
- *LVI IMU*. A design for an inertial measurement unit that costs less than \$150 to build. Complete schematics, board layout files and CAN node interface software are available.
- *CAN Bus Driver*. Modest changes to Arnaud Westenberg's Linux CAN Bus Driver (Section 5.1.4) customize it to our COTS flight computer board and are being contributed back to that project. More importantly, we plan to create and contribute an RTLinux CAN Bus Driver derived from that work.
- *Navigation Algorithms*. The signal processing algorithms for approximating system location (Section 3.1) may be adaptable to other combined inertial and GPS navigation applications.

## 9 Next Steps and Future Work

In late June or early July 2003, we are planning a low-altitude launch to flight test the avionics system to 6 km (20,000 ft) in central Oregon. This test will verify the basic operation of the critical hardware and software systems during flight. Systems to be tested include:

- *Avionics*. The flight computer software including crude navigation algorithms, the ARRL 802.11b

telemetry link, the emergency uplink system, and critical sensors and actuators such as the IMU and recovery system.

- *Launch Tower*. Launch tower computer, launch software, umbilical cord and launch safety systems.
- *Ground Computers*. telemetry display and uplink software.

In September 2003, we are planning to build on the June results and launch to 23 km (75,000 ft) in the Black Rock Desert of Nevada. By the September launch, we hope to:

- Migrate the flight computer software to RTLinux.
- Improve the sensor suite, including a next generation IMU, the first generation of the GPL-GPS, and a magnetometer.
- Improve the navigation algorithms, including real-time GPS-aided inertial navigation routines to calculate position, attitude and trajectory.
- Upgrade the ARRL 802.11b telemetry link to use forward error correction algorithms to improve the quality of data received and provide an ability to confidently reconstruct some amount of lost data.
- Integrate the GPL-GPS receivers into the rocket and base station, creating a local differential GPS system.

After these two launches we will begin a parallel development effort to:

- Develop a steerable hybrid motor system, probably using liquid injection thrust vector control.
- Further tune the navigation algorithms in the avionics system through simulations and experiments with ducted fans.

By late 2004 or early 2005 we hope to begin the integration of these two systems and begin the first of many test flights with an active guidance system in place. Instead of trying to fly a fully actively guided rocket in one step, we plan to continue our careful, incremental approach to development. For example, by increasing the gain of the control system while removing fin area, we can build confidence in the navigation system as the rocket becomes more and more unstable because of the lack of fins.

By late 2005 or early 2006 we hope to have an actively guided, possibly staged, rocket lifting research projects to high altitudes, if not to the edge of space.

## 10 Conclusion

The PSAS software system employs integrated computing, from 8 bit and 32 bit microcontrollers through common PC laptop and server hardware. The system hosts a wide variety of visualization, device control, software development, promotional and coordination roles.



Data flows through real-time serial and parallel buses as well as wireless links, requiring only straightforward programming based on network abstractions. Required tasks range from hard real time signal processing in a preemptive multitasking environment to coordinating a caravan of two dozen people to the Nevada desert. Our budget for development equipment is quite small, and our team of contributors vary widely in skill level and available time.

With minor exceptions, free software has helped us tackle all of these technological and logistical problems. Commercial systems could not have scaled so widely, cost so little, offered so many choices, offered free support and training, and integrated these many domains together so well. Our group could not have grown as quickly, or achieved our current level of sophistication, without free software and open hardware.

## Availability

For complete technical details on our projects, including launch videos, technology overviews, white papers, source code, schematics, and board designs, please visit <http://psas.pdx.edu/>.

## Acknowledgements

Throughout this paper, we have acknowledged the important debt we owe to developers of free software and previous amateur aerospace groups who have made our project possible. We are extremely grateful for the invaluable encouragement and financial support from Portland State University's Computer Science and Electrical and Computer Engineering Departments. We would like to thank the IEEE, AT&T and the NASA/Oregon Space Grant, all of which have provided significant financial support; also the companies which have donated materials and software such as HI-TECH Software, ANSOFT, and Microchip. Finally, we would like to thank Usenix for supporting our efforts to bring free software to the amateur aerospace community, and Keith Packard for providing excellent counsel on our paper.

## References

- [alt] John Coker's Rocket Pages: Altimeter Comparison. Web document. URL <http://www.jcrocket.com/altimeters.shtml> accessed April 2, 2003 07:00 UTC.
- [ama] Amanda, The Advanced Maryland Automatic Network Disk Archiver. Web document. URL <http://www.amanda.org/> accessed April 6, 2003 10:00 UTC.
- [arr] Amateur Radio Relay League High-Speed Digital Networks. Web document. URL <http://www.arrl.org/hsmm/> accessed April 6, 2003 10:00 UTC.
- [C18] HI-TECH Software PICC-18 Compiler. Web document. URL <http://www.htsoft.com> accessed April 6, 2003 23:00 UTC.
- [can] Robert Bosch GmbH CAN home page. Web document. URL <http://www.can.bosch.com/> accessed April 6, 2003 10:00 UTC.
- [CC99] Charles K. Chui and Guanrong Chen. *Kalman Filtering*. Springer-Verlag, 1999.
- [csl] United States Code, Title 49 Transportation, Chapters 701 and 703. Web document. URL <http://uscode.house.gov/title49.htm> accessed April 7, 2003 06:19 UTC.
- [cvs] CVS. Web document. URL <http://www.cvshome.org/> accessed April 6, 2003 09:30 UTC.
- [deb] Debian. Web document. URL <http://www.debian.org/> accessed April 6, 2003 09:30 UTC.
- [FB99] Jay A. Farrell and Matthew Barth. *The Global Positioning System and Inertial Navigation*. McGraw-Hill, 1999.
- [fsm] FSMLabs - The RTLinux Company. Web document. URL <http://www.fsmlabs.com/> accessed April 6, 2003 10:00 UTC.
- [Gal95] Bill O. Gallmeister. *POSIX.4: Programming for the real world*. O'Reilly and Associates, Inc., 1995.
- [gpsa] GPL-GPS home page. Web document. URL <http://gps.psas.pdx.edu/> accessed April 6, 2003 23:00 UTC.
- [gpsb] Zarlink Semiconductor GP4020 GPS Receiver Baseband Processor. Web document. URL [http://products.zarlink.com/product\\_profiles/GP4020.htm](http://products.zarlink.com/product_profiles/GP4020.htm) accessed April 6, 2003 10:00 UTC.
- [ins] Ariane 5 - Flight 501 Failure; Report by the Inquiry Board, European Space Agency. Web document. URL <http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html> accessed April 7, 2003 03:00 UTC.
- [jfr] JFreeChart. Freely available source code. URL [www.jfree.org/jfreechart/index.html](http://www.jfree.org/jfreechart/index.html) accessed April 7, 2003 00:48 UTC.
- [kel] OpenSource GPS Project. Web document. URL <http://home.earthlink.net/>

- ~cwkelley/ accessed April 6, 2003 23:00 UTC.
- [Lab99] Jean J. Labrosse. *MicroC/OS-II: The Real Time Kernel*. Miller Freeman, 1999.
- [lcb] Linux CAN-bus Driver for the Intel(c) 82527 and Philips sjal000 controllers. Web document. URL <http://home.wanadoo.nl/arnaud/>, accessed April 7, 2003 07:21 UTC.
- [mai] Mailman, the GNU Mailing List Manager. Web document. URL <http://www.list.org/> accessed April 6, 2003 09:30 UTC.
- [maj] Majordomo. Web document. URL <http://www.greatcircle.com/majordomo/> accessed April 6, 2003 09:30 UTC.
- [mar] MARS Amateur Rocketry Group. Web document. URL <http://www.mars.org.uk/> accessed April 6, 2003 23:00 UTC.
- [Mas03] Anthony J. Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2003.
- [pic] Microchip, inc. Web document. URL <http://www.microchip.com/> accessed April 6, 2003 23:00 UTC.
- [rrs] Reaction Research Society Space Shot. Web document. URL [http://www.rrs.org/Projects/Launches/Space\\_Shot/space\\_shot.html](http://www.rrs.org/Projects/Launches/Space_Shot/space_shot.html) accessed April 2, 2003 07:00 UTC.
- [SB00] George P. Sutton and Oscar Biblar. *Elements of Rocket Propulsion*. Wiley-Interscience, December 2000.
- [sls] Spacelawstation.com: U.S. Space Law. Web document. URL <http://www.spacelawstation.com/uslaw.html> accessed April 7, 2003 06:20 UTC.
- [ssh] OpenSSH. Web document. URL <http://www.openssh.org/> accessed April 6, 2003 09:30 UTC.
- [Tri] Tripoli Rocketry Association. Motor size classifications. Web document. URL [http://www.tripoli.org/motors/motor\\_classes.html](http://www.tripoli.org/motors/motor_classes.html) accessed April 3, 2003 19:00 UTC.
- [twi] TWiki - A Web Based Collaboration Platform. Web document. URL <http://www.twiki.org/> accessed April 6, 2003 09:30 UTC.
- [un] United Nations Office for Outer Space Affairs; International Space Law. Web document. URL <http://www.oosa.unvienna.org/SpaceLaw/spacelaw.htm> accessed April 7, 2003 02:00 UTC.
- [Wer92] James R. Wertz, editor. *Space Mission Analysis and Design*. Microcosm, Inc., October 1992.
- [Win83] Frank H. Winter. *Prelude to the Space Age: The Rocket Societies 1924-1940*. Smithsonian Institution Press, 1983.

# POSIX Access Control Lists on Linux

Andreas Grünbacher  
*SuSE Labs, SuSE Linux AG*  
*Nuremberg, Germany*  
agruen@suse.de

## Abstract

This paper discusses file system Access Control Lists as implemented in several UNIX-like operating systems. After recapitulating the concepts of these Access Control Lists that never formally became a POSIX standard, we focus on the different aspects of implementation and use on Linux.

## 1 Introduction

Traditionally, systems that support the POSIX (Portable Operating System Interface) family of standards [2, 11] share a simple yet powerful file system permission model: Every file system object is associated with three sets of permissions that define access for the owner, the owning group, and for others. Each set may contain Read (r), Write (w), and Execute (x) permissions. This scheme is implemented using only nine bits for each object. In addition to these nine bits, the Set User Id, Set Group Id, and Sticky bits are used for a number of special cases. Many introductory and advanced texts on the UNIX operating system describe this model [19].

Although the traditional model is extremely simple, it is sufficient for implementing the permission scenarios that usually occur on UNIX systems. System administrators have also found several workarounds for the model's limitations. Some of these workarounds require nonobvious group setups that may not reflect organizational structures. Only the root user can create groups or change group membership. Set-user-ID root utilities may allow ordinary users to perform some administrative tasks, but bugs in such utilities can easily lead to compromised systems. Some applications like FTP daemons implement their own extensions to the file system permission model [15]. The price of playing tricks with permissions is an increase in complexity of the system configuration. Understanding and maintaining the integrity of systems becomes more difficult.

Engineers have long recognized the deficiencies of the traditional permission model and have started to think about alternatives. This has eventually resulted in a number of Access Control List (ACL) implementations on UNIX, which are only compatible among each other to a limited degree.

This paper gives an overview of the most successful ACL scheme for UNIX-like systems that has resulted from the POSIX 1003.1e/1003.2c working group.

After briefly describing the concepts, some examples of how these are used are given for better understanding. Following that, the paper discusses Extended Attributes, the abstraction layer upon which ACLs are based on Linux. The rest of the paper deals with implementation, performance, interoperability, application support, and system maintenance aspects of ACLs.

The author was involved in the design and implementation of extended attributes and ACLs on Linux, which covered the user space tools and the kernel implementation for Ext2 and Ext3, Linux's most prominent file systems. Parts of the design of the system call interface are attributed to Silicon Graphics's Linux XFS project, particularly to Nathan Scott.

## 2 The POSIX 1003.1e/1003.2c Working Group

A need for standardizing other security relevant areas in addition to just ACLs was also perceived, so eventually a working group was formed to define security extensions within the POSIX 1003.1 family of standards. The document numbers 1003.1e (System Application Programming Interface) and 1003.2c (Shell and Utilities) were assigned for the working group's specifications. These documents are referred to as POSIX.1e in the remainder of this paper. The working group was focusing on the following extensions to POSIX.1: Access Control Lists (ACL), Audit, Capability, Mandatory Access Control (MAC), and Information Labeling.

Unfortunately, it eventually turned out that standardizing all these diverse areas was too ambitious a goal. In January 1998, sponsorship for 1003.1e and 1003.2c was withdrawn. While some parts of the documents produced by the working group until then were already of high quality, the overall works were not ready for publication as standards. It was decided that draft 17, the last version of the documents the working group had produced, should be made available to the public. Today, these documents can be found at Winfried Trümper's Web site [27].

Several UNIX system vendors have implemented var-

ious parts of the security extensions, augmented by vendor-specific extensions. The resulting versions of their operating systems have often been labeled “trusted” operating systems, e.g., Trusted Solaris, Trusted Irix, Trusted AIX. Some of these “trusted” features have later been incorporated into the vendors’ main operating systems.

ACLs are supported on different file system types on almost all UNIX-like systems nowadays. Some of these implementations are compatible with draft 17 of the specification, while others are based on older drafts. Unfortunately, this has resulted in a number of subtle differences among the different implementations.

The TrustedBSD project (<http://www.trustedbsd.org/>) lead by Robert Watson has implemented versions of the ACL, Capabilities, MAC, and Audit parts of POSIX.1e for FreeBSD. The ACL and MAC implementations appear in FreeBSD-RELEASE as of January, 2003. The MAC implementation is still considered experimental.

### 3 Status of ACLs on Linux

Patches that implement POSIX 1003.1e draft 17 ACLs have been available for various versions of Linux for several years now. They were added to version 2.5.46 of the Linux kernel in November 2002. Current Linux distributions are still based on the 2.4.x stable kernels series. SuSE and the United Linux consortium have integrated the 2.4 kernel ACL patches earlier than others, so their current products offer the most complete ACL support available for Linux to date. Other vendors apparently are still reluctant to make that important change, but experimental versions are expected to be available later this year.

The Linux *getfacl* and *setfacl* command line utilities do not strictly follow POSIX 1003.2c draft 17, which shows mostly in the way they handle default ACLs. See section 6.

At the time of this writing, ACL support on Linux is available for the Ext2, Ext3, IBM JFS, ReiserFS, and SGI XFS file systems. Solaris-compatible ACL support for NFS version 3 exists since March 3, 2003.

### 4 How ACLs Work

The traditional POSIX file system object permission model defines three classes of users called owner, group, and other. Each of these classes is associated with a set of permissions. The permissions defined are read (r), write (w), and execute (x). In this model, the *owner class* permissions define the access privileges of the file owner, the *group class* permissions define the access privileges of the owning group, and the *other class* permissions define the access privileges of all users that are not in one of these two classes. The *ls -l* command displays the owner, group, and other class permissions in

Entry type	Text form
Owner	user : rwx
Named user	user : name : rwx
Owning group	group : rwx
Named group	group : name : rwx
Mask	mask : rwx
Others	other : rwx

Table 1: Types of ACL Entries

the first column of its output (e.g., “- rw- r- - - -” for a regular file with read and write access for the owner class, read access for the group class, and no access for others).

An ACL consists of a set of entries. The permissions of each file system object have an ACL representation, even in the minimal, POSIX.1-only case. Each of the three classes of users is represented by an ACL entry. Permissions for additional users or groups occupy additional ACL entries.

Table 1 shows the defined entry types and their text forms. Each of these entries consists of a type, a qualifier that specifies to which user or group the entry applies, and a set of permissions. The qualifier is undefined for entries that require no qualification.

ACLs equivalent with the file mode permission bits are called *minimal* ACLs. They have three ACL entries. ACLs with more than the three entries are called *extended* ACLs. Extended ACLs also contain a mask entry and may contain any number of named user and named group entries.

These named group and named user entries are assigned to the *group class*, which already contains the owning group entry. Different from the POSIX.1 permission model, the group class may now contain ACL entries with different permission sets, so the group class permissions alone are no longer sufficient to represent all the detailed permissions of all ACL entries it contains. Therefore, the meaning of the group class permissions is redefined: under their new semantics, they represent an upper bound of the permissions that any entry in the group class will grant.

This upper bound property ensures that POSIX.1 applications that are unaware of ACLs will not suddenly and unexpectedly start to grant additional permissions once ACLs are supported.

In minimal ACLs, the group class permissions are identical to the owning group permissions. In extended ACLs, the group class may contain entries for additional users or groups. This results in a problem: some of these additional entries may contain permissions that are not contained in the owning group entry, so the owning group entry permissions may differ from the group class permissions.

This problem is solved by the virtue of the mask entry.



With minimal ACLs, the group class permissions map to the owning group entry permissions. With extended ACLs, the group class permissions map to the mask entry permissions, whereas the owning group entry still defines the owning group permissions. The mapping of the group class permissions is no longer constant. Figure 1 shows these two cases.

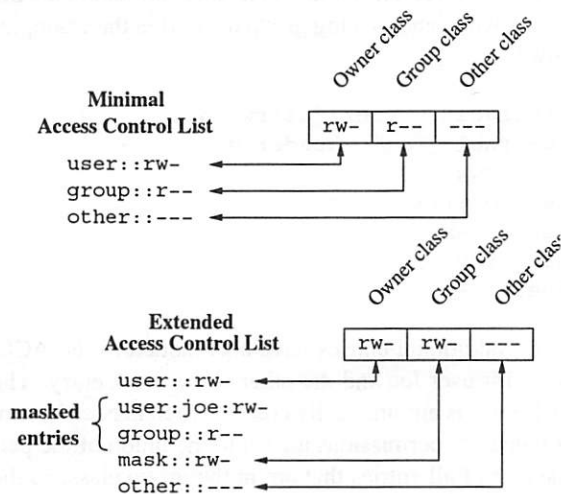


Figure 1: Mapping between ACL Entries and File Mode Permission Bits

When an application changes any of the owner, group, or other class permissions (e.g., via the *chmod* command), the corresponding ACL entry changes as well. Likewise, when an application changes the permissions of an ACL entry that maps to one of the user classes, the permissions of the class change.

The group class permissions represent the upper bound of the permissions granted by any entry in the group class. With minimal ACLs this is trivially the case. With extended ACLs, this is implemented by masking permissions (hence the name of the mask entry): permissions in entries that are a member of the group class which are also present in the mask entry are effective. Permissions that are absent in the mask entry are masked and thus do not take effect. See Table 2.

Entry type	Text form	Permissions
Named user	user:joe:r-x	r-x
Mask	mask::rw-	rw-
<b>Effective permissions</b>		<b>r--</b>

Table 2: Masking of Permissions

The owner and other entries are not in the group class. Their permissions are always effective and never masked.

So far we have only looked at ACLs that define the cur-

rent access permissions of file system objects. This type is called *access* ACL. A second type called *default* ACL is also defined. They define the permissions a file system object inherits from its parent directory at the time of its creation. Only directories can be associated with default ACLs. Default ACLs for non-directories would be of no use, because no other file system objects can be created inside non-directories. Default ACLs play no direct role in access checks.

When a directory is created inside a directory that has a default ACL, the new directory inherits the parent directory's default ACL both as its access ACL and default ACL. Objects that are not directories inherit the default ACL of the parent directory as their access ACL only.

The permissions of inherited access ACLs are further modified by the *mode* parameter that each system call creating file system objects has. The *mode* parameter contains nine permission bits that stand for the permissions of the owner, group, and other class permissions. The effective permissions of each class are set to the intersection of the permissions defined for this class in the ACL and specified in the *mode* parameter.

If the parent directory has no default ACL, the permissions of the new file are determined as defined in POSIX.1. The effective permissions are set to the permissions defined in the *mode* parameter, minus the permissions set in the current *umask*.

The umask has no effect if a default ACL exists.

## 4.1 Access Check Algorithm

A process requests access to a file system object. Two steps are performed. Step one selects the ACL entry that most closely matches the requesting process. The ACL entries are looked at in the following order: owner, named users, (owning or named) groups, others. Only a single entry determines access. Step two checks if the matching entry contains sufficient permissions.

A process can be a member in more than one group, so more than one group entry can match. If any of these matching group entries contain the requested permissions, one that contains the requested permissions is picked (the result is the same no matter which entry is picked). If none of the matching group entries contains the requested permissions, access will be denied no matter which entry is picked.

The access check algorithm can be described in pseudo-code as follows.

```

if the user ID of the process is the owner, the owner
    entry determines access
else if the user ID of the process matches the qualifier in
    one of the named user entries, this entry determines
    access
else if one of the group IDs of the process matches the
    owning group and the owning group entry contains
  
```

the requested permissions, this entry determines access

**else if** one of the group IDs of the process matches the qualifier of one of the named group entries and this entry contains the requested permissions, this entry determines access

**else if** one of the group IDs of the process matches the owning group or any of the named group entries, but neither the owning group entry nor any of the matching named group entries contains the requested permissions, this determines that access is denied

**else** the other entry determines access.

**If** the matching entry resulting from this selection is the owner or other entry and it contains the requested permissions, access is granted

**else if** the matching entry is a named user, owning group, or named group entry and this entry contains the requested permissions and the mask entry also contains the requested permissions (or there is no mask entry), access is granted

**else** access is denied.

## 5 Access ACL Example

Let us start by creating a directory and checking its permissions. The *umask* determines which permissions will be masked off when the directory is created. A *umask* of 027 (octal) disables write access for the owning group and read, write, and execute access for others.

```
$ umask 027
$ mkdir dir
$ ls -dl dir
drwxr-x--- ... agruen suse ... dir
```

The first character *ls* prints represents the file type (d for directory). The string "rwx r-x ---" represents the resulting permissions for the new directory: read, write, and execute access for the owner and read and execute access for the owning group. The dots in the output of *ls* stand for text that is not relevant here and has been removed.

These base permissions have an equivalent representation as an ACL. ACLs are displayed using the *getfacl* command.

```
$ getfacl dir
# file: dir
# owner: agruen
# group: suse
user::rwx
group::r-x
other::---
```

The first three lines of output contain the file name, owner, and owning group of the file as comments. Each

of the following lines contains an ACL entry for one of the three classes of users: owner, group, and other.

The next example grants read, write, and execute access to user Joe in addition to the existing permissions. For that, the *-m* (modify) argument of *setfacl* is used. The resulting ACL is again shown using the *getfacl* command. The *-omit-header* option to *getfacl* suppresses the three-line comment header containing the file name, owner, and owning group to shorten the examples shown.

```
$ setfacl -m user:joe:rwx dir
$ getfacl --omit-header dir
user::rwx
user:joe:rwx
group::r-x
mask::rwx
other::---
```

Two additional entries have been added to the ACL: one is for user Joe and the other is the mask entry. The mask entry is automatically created when needed but not provided. Its permissions are set to the union of the permissions of all entries that are in the group class, so the mask entry does not mask any permissions.

The mask entry now maps to the group class permissions. The output of *ls* changes as shown next.

```
$ ls -dl dir
drwxrwx---+ ... agruen suse ... dir
```

An additional "+" character is displayed after the permissions of all files that have extended ACLs. This seems like an odd change, but in fact POSIX.1 allocates this character position to the optional alternate access method flag, which happens to default to a space character if no alternate access methods are in use.

The permissions of the group class permissions include write access. Traditionally such file permission bits would indicate write access for the owning group. With ACLs, the effective permissions of the owning group are defined as the intersection of the permissions of the owning group and mask entries. The effective permissions of the owning group in the example are still r-x, the same permissions as before creating additional ACL entries with *setfacl*.

The group class permissions can be modified using the *setfacl* or *chmod* command. If no mask entry exists, *chmod* modifies the permissions of the owning group entry as it does traditionally. The next example removes write access from the group class and checks what happens.

```
$ chmod g-w dir
$ ls -dl dir
drwxr-x---+ ... agruen suse ... dir
```

```
$ getfacl --omit-header dir
user::rwx
user:joe:rwx          #effective:r-x
group::r-x
mask::r-x
other::---
```

As shown, if an ACL entry contains permissions that are disabled by the mask entry, *getfacl* adds a comment that shows the effective set of permissions granted by that entry. Had the owning group entry had write access, there would have been a similar comment for that entry. Now see what happens if write access is given to the group class again.

```
$ chmod g+w dir
$ ls -dl dir
drwxrwx---+ ... agruen suse ... dir
$ getfacl --omit-header dir
user::rwx
user:joe:rwx
group::r-x
mask::rwx
other::---
```

After adding the write permission to the group class, the ACL defines the same permissions as before taking the permission away. The *chmod* command has a nondestructive effect on the access permissions. This preservation of permissions is an important feature of POSIX.1e ACLs.

## 6 Default ACL Example

In the following example, we add a default ACL to the directory. Then we check what *getfacl* shows.

```
$ setfacl -d -m group:toolies:r-x dir
$ getfacl --omit-header dir
user::rwx
user:joe:rwx
group::r-x
mask::rwx
other::---
default:user::rwx
default:group::r-x
default:group:toolies:r-x
default:mask::r-x
default:other::---
```

Following the access ACL, the default ACL is printed with each entry prefixed with “default:”. This output format is an extension to POSIX.1e that is found on Solaris and Linux. A strict implementation of POSIX 1003.2c would only show the access ACL. The default ACL would be shown with the *-d* option to *getfacl*.

We have only specified an ACL entry for the *toolies* group in the *setfacl* command. The other entries required

for a complete ACL have automatically been copied from the access ACL to the default ACL. This is a Linux-specific extension; on other systems all entries may need to be specified explicitly.

The default ACL contains no entry for Joe, so Joe will not have access (except possibly through group membership or the other class permissions).

A subdirectory inherits ACLs as shown next. Unless otherwise specified, the *mkdir* command uses a value of 0777 as the *mode* parameter to the *mkdir* system call, which it uses for creating the new directory. Observe that both the access and the default ACL contain the same entries.

```
$ mkdir dir/subdir
$ getfacl --omit-header dir/subdir
user::rwx
group::r-x
group:toolies:r-x
mask::r-x
other::---
default:user::rwx
default:group::r-x
default:group:toolies:r-x
default:mask::r-x
default:other::---
```

Files created inside *dir* inherit their permissions as shown next. The *touch* command passes a *mode* value of 0666 to the kernel for creating the file.

All permissions not included in the *mode* parameter are removed from the corresponding ACL entries. The same has happened in the previous example, but there was no noticeable effect because the value 0777 used for the *mode* parameter represents a full set of permissions.

```
$ touch dir/file
$ ls -l dir/file
-rw-r-----+ ... agruen suse ... dir/file
$ getfacl --omit-header dir/file
user::rw-
group::r-x          #effective:r--
group:toolies:r-x   #effective:r--
mask::r--
other::---
```

No permissions have been removed from ACL entries in the group class; instead they are merely masked and thus made ineffective. This ensures that traditional tools like compilers will interact well with ACLs. They can create files with restricted permissions and mark the files executable later. The mask mechanism will cause the right users and groups to end up with the expected permissions.

## 7 Extended Attributes

In this section we begin detailing the implementation of ACLs in Linux.

ACLs are pieces of information of variable length that are associated with file system objects. Dedicated strategies for storing ACLs on file systems might be devised, as Solaris does on the UFS file system [13]. Each inode on a UFS file system has a field called *i\_shadow*. If an inode has an ACL, this field points to a *shadow inode*. On the file system, shadow inodes are used like regular files. Each shadow inode stores an ACL in its data blocks. Multiple files with the same ACL may point to the same shadow inode.

Because other kernel and user space extensions in addition to ACLs benefit from being able to associate pieces of information with files, Linux and most other UNIX-like operating systems implement a more general mechanism called Extended Attributes (EAs). On these systems, ACLs are implemented as EAs.

Extended attributes are name and value pairs associated permanently with file system objects, similar to the environment variables of a process. The EA system calls used as the interface between user space and the kernel copy the attribute names and values between the user and kernel address spaces. The Linux `attr(5)` manual page contains a more complete description of EAs as found on Linux. A paper by Robert Watson discussing supporting infrastructure for security extensions in FreeBSD contains a comparison of different EA implementations on different systems [25].

Other operating systems, such as Sun Solaris, Apple MacOS, and Microsoft Windows, allow multiple streams (or forks) of information to be associated with a single file. These streams support the usual file semantics. After obtaining a handle on the stream, it is possible to access the streams' contents using ordinary file operations like read and write. Confusingly, on Solaris these streams are called extended attributes as well. The EAs on Linux and several other UNIX-like operating systems have nothing to do with these streams. The more limited EA interface offers several advantages. They are easier to implement, EA operations are inherently atomic, and the stateless interface does not suffer from overheads caused by obtaining and releasing file handles. Efficiency is important for frequently accessed objects like ACLs.

At the file system level, the obvious and straightforward approach to implement EAs is to create an additional directory for each file system object that has EAs and to create one file for each extended attribute that has the attribute's name and contains the attribute's value. Because on most file systems allocating an additional directory plus one or more files requires several disk blocks, such a simple implementation would consume

a lot of space, and it would not perform very well because of the time needed to access all these disk blocks. Therefore, most file systems use different mechanisms for storing EAs.

### 7.1 Ext2 and Ext3

As described in the Linux kernel sources, each inode has a field that is called *i\_file\_acl* for historic reasons. If this field is not zero, it contains the number of the file system block on which the EAs associated with this inode are stored. This block contains both the names and values of all EAs associated with the inode. All EAs of an inode must fit on the same EA block.

For improved efficiency, multiple inodes with identical sets of EAs may point to the same EA block. The number of inodes referring to an EA block are tracked by a reference count in the EA block. EA block sharing is transparent for the user: Ext3 keeps an LRU list of recently accessed EA blocks and a table that has two indices (implemented as hash tables of double linked lists). One index is by block number. The other is by a checksum of the block's contents. Blocks that contain the same data with which a new inode shall be associated are reused until the block's reference count reaches an upper limit of 1024. This limits the damage a single disk block failure may cause. When an inode refers to a shared EA block and that inode's EAs are changed, a copy-on-write mechanism is used, unless another cached EA block already contains the same set of attributes, in which case that block is used.

The current implementation requires all EAs of an inode to fit on a single disk block, which is 1, 2, or 4 KiB. This also determines the maximum size of individual attributes.

If the sets of EAs tend to be unique among inodes, no sharing is possible and the time spent checking for potential sharing is wasted. If each inode has a unique set of EAs, each of these sets will be stored on a separate disk block, which can lead to a lot of slack space. The extreme case is applications that need to store unique EAs for each inode. Fortunately for many common workloads, the EA sharing mechanism is highly effective.

Alternative designs with fewer limitations have been proposed [5], but it seems that they are not easy to actually implement. No alternatives to the existing scheme exist so far.

### 7.2 JFS

JFS stores all EAs of an inode in a consecutive range of blocks on the file system (i.e., in an extent) [3]. The extended attribute name and value pairs are stored consecutively in this extent. If the entire EAs are small enough, they are stored entirely within the inode to which they



belong.

JFS does not implement an EA sharing mechanism. It does not have the one-disk-block limitation of Ext2 and Ext3. The size of individual attributes is limited to 64 KiB.

### 7.3 XFS

Of the file systems currently supported in Linux, XFS uses the most elaborate scheme for storing extended attributes [1]. Small sets of EAs are stored directly in inodes, medium-sized sets are stored on leaf blocks of B+ trees, and large sets of EAs are stored in full B+ trees. This results in performance characteristics similar to directories on XFS: although rarely needed, very large numbers of EAs can be stored efficiently.

XFS has a configurable inode size that is determined at file system create time. The minimum size is 256 bytes, which is also the default. The maximum size is one half of the file system block size. In the minimum case, the inodes are too small to hold ACLs, so they will be stored externally. If the inode size is increased, ACLs will fit directly in the inode. Since inodes and their ACLs are often accessed within a short period of time, this results in faster access checks, but also wastes more disk space.

XFS does not have an attribute sharing mechanism. The size of individual attributes is limited to 64 KiB.

### 7.4 ReiserFS

ReiserFS supports tail merging of files, which means that several files can share the same disk block for storing their data. This makes the file system very efficient for many small files. One potential drawback is that tail merging can consume a noticeable amount of CPU time.

Since ReiserFS is so good at handling small files, EAs can directly use this mechanism. For each file that has EAs, a directory with a name derived from a unique inode identifier is created inside a special directory. The special directory is usually hidden from the file system namespace. Inside the inode specific directory, each EA is stored as a separate file. The file name equals the attribute name. The file's contents are the attribute value.

ReiserFS does not implement an attribute sharing mechanism, but such an extension will possibly be implemented in the future. Sharing could even be implemented on a per-attribute bases, so the result would be a highly efficient and flexible solution. The size of individual attributes is limited to 64 KiB.

## 8 ACL Implementations

An interesting design decision is how ACLs should be passed between user space and the kernel, and inside the kernel, between the virtual file system (VFS) and the low-level file system layer. FreeBSD, Solaris, Irix,

and HP-UX all have separate ACL system calls [9, 17, 21, 23].

Linux does not have ACL system calls. Instead, ACLs are passed between the kernel and user space as EAs. This reduces the number of system interfaces, but with the same number of end operations. While the ACL system calls provide a more explicit system interface, the EA interface is easier to adapt to future requirements, such as non-numerical identifiers for users and groups in ACL entries.

The rationale for using separate ACL system calls in FreeBSD was that some file systems support EAs but not ACLs, and some file systems support ACLs but not EAs, so EAs are treated as pure binary data. EAs and ACLs only become related inside a file system. [24, 26].

The rationale for the Linux design was to provide access to all meta data pertinent to a file system object through the same interface. Different classes of attributes that are recognized by name are reserved for system objects such as ACLs. The attribute names "system.posix\_acl\_access" and "system.posix\_acl\_default" are used for the access and default ACL of a file, respectively. The ACL attribute values are in a canonical, architecture-independent binary format. File systems that do not implement ACLs but do implement EAs, or ones that implement ACLs as something other than EAs, need to recognize the relevant attribute names.

While it is possible to manipulate ACLs directly as EAs, at the application level this is usually not done: since the EA system calls are Linux-specific, such applications would not be portable. Other systems support similar EA mechanisms, but with different system call interfaces. Applications that want to use POSIX.1 ACLs in a portable way are expected to use the *libacl* library, which implements the ACL-specific functions of POSIX.1e draft 17.

The access ACL of a file system object is accessed for every access decision that involves that object. Access checking is performed on the whole path from the namespace root to the file in question. It is important that ACL access checks are efficient. To avoid frequently looking up ACL attributes and converting them from the machine-independent attribute representation to a machine-specific representation, the Ext2, Ext3, JFS, and ReiserFS implementations cache the machine-specific ACL representations. This is done in addition to the normal file system caching mechanisms, which use either the page cache, the buffer cache, or both. XFS does not use this additional layer of caching.

Most UNIX-like systems that support ACLs limit the number of ACL entries allowed to some reasonable number. Table 3 shows the limits on Linux.

ACLs with a high number of ACL entries tend to become more difficult to manage. More than a handful of

File system	Max. entries
XFS	25
Ext2, Ext3	32
ReiserFS, JFS	8191

Table 3: Maximum Number of Supported ACL Entries

ACL entries are usually an indication of bad application design. In most such cases, it makes more sense to make better use of groups instead of bloating ACLs.

The ReiserFS and JFS implementations define no limit on the number of ACL entries, so a limit is only imposed by the maximum size of EA values. The current EA size limit is 64 KiB, or 8191 ACL entries, which is too high for ACLs in practice: besides being impractical to work with, the time it would take to check access in such huge ACLs may be prohibitive.

## 9 Compatibility

An important aspect of introducing new file system features is how systems are upgraded, and how systems that do not support the new features are affected. File system formats evolve slowly. File systems are expected to continue to work with older versions of the kernel. In some situations, like in multiple boot environments or when booting from a rescue system, it may be necessary or preferable to use a kernel that does not have EA support.

All file systems that support ACLs on Linux are either inherently aware of EAs, or are upgraded to support EAs automatically, without user intervention. Depending on the file system, this is either done during mounting or when first using extended attributes.

On all file systems discussed in this paper, when using a kernel that does not support EAs on a file system with EAs, the EAs will be ignored. On ReiserFS, EA-aware kernels actively hide the system directory that contains the EAs, so in an EA-unaware kernel this directory becomes visible. It is still protected from ordinary users through file permissions.

Working with EA file systems with EA-unaware kernels will still lead to inconsistencies when files are deleted that have EAs. In that case, the EAs will not get removed and a disk space leak will result. At least on Ext2 and Ext3, such inconsistencies can be cleaned up later by running the file system checker.

ACL-unaware kernels will only see the traditional file permission bits and will not be able to check permissions defined in ACLs. The ACL inheritance algorithm will not work.

## 10 EA and ACL Performance

Since ACLs define a more sophisticated discretionary access control mechanism, they have an influence on all access decisions for file system objects. It is interesting

to compare the time it takes to perform an access decision with and without ACLs.

Measurements were performed on a PC running SuSE Linux 8.2, with the SuSE 2.4.20 kernel. The machine has an AMD Athlon processor clocked at 1.1 GHz and 512 MiB of RAM. The disk used was a 30 GB IBM Ultra ATA 100 hard drive with 7200 RPM, an average seek time of 9.8 ms, and 2 MiB of on-disk cache. The Ext2, Ext3, Reiserfs, and JFS file systems were created with default options on an 8 GiB partition. On XFS, to compare EAs that are stored in inodes and EAs that are stored externally, file systems with inode sizes of 256 bytes and 512 bytes were used. These file systems are labeled *XFS-256* and *XFS-512*, respectively.

Table 4 compares the times required for the initial access check to a file with and without an ACL after system restart. To exclude the time for loading the file's inode into the cache, a *stat* system call was performed before checking access. The time taken for the *stat* system call is not shown. The first access to the access ACL of a file may require one or more disk accesses, which are several orders of magnitude slower than accessing the cache. The actual times these disk accesses take vary widely depend on the disk speed and on the relative locations of the disk blocks that are accessed. The function used for measuring time has a resolution of 1 microsecond. In the ACL case, the file that is checked has a five-entry access ACL.

	Without ACL	With ACL
Ext2	9	1743
Ext3	10	3804
ReiserFS	9	6165
XFS-256	14	7531
XFS-512	14	14
JFS	13	13

Table 4: Microseconds for Initially Accessing a File After System Restart, with and without ACLs

XFS with 512-byte (or larger) inodes and JFS store the ACLs directly in the inodes. Therefore, no additional disk accesses are needed for retrieving the ACLs.

After the first repetition, all information is fully cached. Figures 2–4 show micro-benchmarks of basic operations in this state. Each test repeats the same operation many times and averages the total time spent over the number of repetitions. In all configurations except XFS-256 with ACLs, the time per access check drops to around 1–2 microseconds, as the leftmost measurements in Figures 3 and 4 show.

Figure 2 compares the speed of various system calls. The *getpid* system call is included to show the overhead of switching between the user and kernel address spaces. The *ls -l* command indicates in its output if a file has an

extended ACL. Internally, it uses the `acl_extended_file` function from the `libacl` library. This function is almost as fast as the `stat` system call, which `ls -l` also calls for each file, so the additional overhead is small. For comparison, the `acl_get_file` measurement shows the time it takes to retrieve a five-entry ACL.

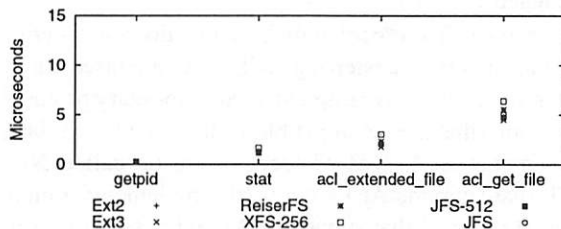


Figure 2: Various System Calls and Library Functions

Figure 3 shows the time taken for one access system call, depending on the number of directory levels in the pathname argument. Figure 4 shows the performance of the same operation with a five-entry access ACL on each directory. XFS's overhead for converting the ACL from its EA representation to its in-memory representation results in a noticeable difference, particularly with 256 byte inodes.

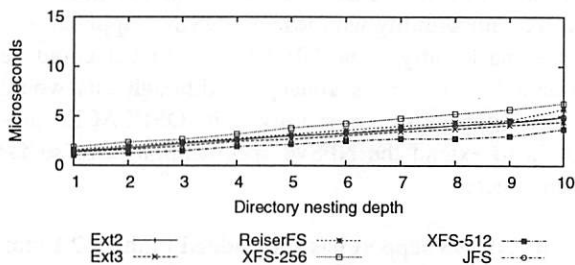


Figure 3: The Access System Call without ACLs

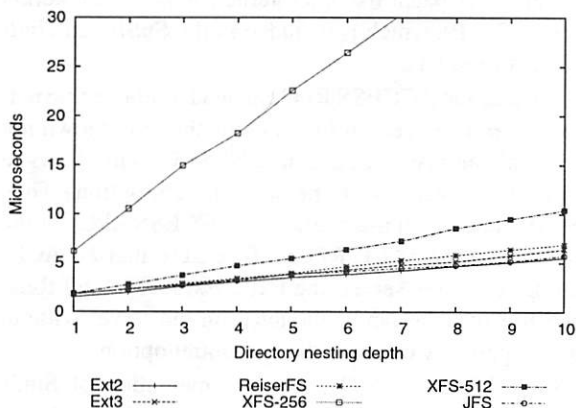


Figure 4: The Access System Call with ACLs

Tables 5 and 6 show the overhead involved when

copying files from one file system to another. All the files that are copied have ACLs, but no additional EAs. Figure 5 shows the distribution of file sizes in the sample data sets. The tests show the time taken from starting the `cp` command to the return from the `sync` command, which we start immediately after the `cp` command. The `sync` command ensures that all files are immediately written. The first series uses a version of `cp` that does not support EAs or ACLs. The second series uses a version of `cp` that supports both EAs and ACLs. In both of the benchmarks, the source file system type is held constant, while the destination file system type is changed.

The sample size for the benchmark in Table 5 is small enough for all files to fit in main memory. The time for first reading the data into memory is not included in the results. These tests were repeated five times. The results show the median and the standard deviation of the results. The sample size for the benchmark shown in Table 6 is larger than either main memory or the file system journals. These tests were only executed once.

File system	Without EAs or ACLs		With ACLs		Overhead (%)
Ext2	18.3	0.2	18.8	0.2	+3
Ext3	22.0	2.4	22.7	0.5	+3
ReiserFS	9.0	0.1	12.8	0.1	+42
XFS-256	19.0	0.2	34.1	0.2	+80
XFS-512	20.1	0.4	21.4	0.2	+7
JFS	38.2	0.6	36.5	0.2	4

Table 5: Seconds for Copying Files From Memory to a File System (11351 files, 608 directories, total file size = 137 MiB)

File system	Without EAs or ACLs		With ACLs		Overhead (%)
Ext2	595		578		3
Ext3	613		623		+2
ReiserFS	518		538		+4
XFS-256	547		641		+17
XFS-512	549		566		+3
JFS	654		590		11

Table 6: Seconds for Copying Files Between File Systems (96183 files, 6323 directories, total file size = 2.8 GiB)

Note that both benchmarks use ACLs excessively, which is a worst-case scenario. The overheads for real workloads should be much smaller.

It can be observed that the overhead of ACLs varies widely among the supported file systems. The differences show more when the I/O load is low, and get smaller as the I/O load rises. For ACLs, the Ext2 and Ext3 implementations have little overhead. ReiserFS EAs have a relatively high overhead. This may improve if attribute sharing is implemented. For XFS, increas-



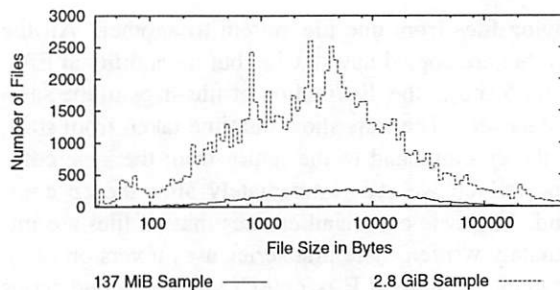


Figure 5: Distribution of File Sizes for Samples of Tables 5 and 6

ing the inode size so that ACLs can be stored directly in inodes makes a big difference.

It is unclear why JFS appears to be faster when copying ACLs than when not copying ACLs.

## 11 NFS and ACLs

Full ACL support over NFS requires two things: First, a mechanism so that all access decisions are performed in a way that honors the ACLs. Second, extensions to the NFS protocol for manipulating ACLs on remote mounted file systems.

The NFS protocol performs client-side caching to improve efficiency. In version 2 of the protocol, decisions as to who gets read access to locally cached data are performed on the client. These decisions are made under the assumption that the file mode permissions bits and the IDs of the owner and owning group are sufficient to do that. This assumption is obviously wrong if an extended permission scheme like POSIX ACLs is used on the server.

Because NFSv2 clients perform some access decisions locally, they will incorrectly grant read access to file and directory contents cached on the client to users who are a member in the owning group in two cases. First, if the group class permissions include read access, but the owning group does not have read access. Second, if the owning group does have read access, but a named user entry for that user exists that does not allow read access. Both situations are rare. Workarounds exist that reduce the permissions on the server side so that clients only see a safe subset of the real permissions [7, 10]. No anomalies exist for users who are not a member in the owning group.

There are two ways to solve this problem. One is to extend the access check algorithm used on the client. The other is to delegate access decisions to the server and possibly cache those decisions for a defined period of time on the client. The first solution would probably scale better to a high number of readers on the client side, as long as the server and all clients can agree on the access check algorithms use. Unfortunately, this ap-

proach falls apart as soon as servers implement different permission schemes.

Version 3 of the NFS protocol therefore defines a new remote procedure call (RPC) called ACCESS for delegating access decisions to the server. This RPC is similar to the access system call. NFSv3 clients are expected to use this RPC for determining to whom to grant access to cached contents.

The NFSv3 protocol unfortunately does not define mechanisms for transferring ACLs. As a consequence, different vendors have implemented proprietary protocol extensions that are incompatible with each other. Solaris implements an NFSv3 protocol extension called NFS ACL that supports ACLs only. Irix implements a more general protocol that supports EAs and passes ACLs as special EAs.

NFSv4 defines the structure and semantics of its own kind of ACLs, along with RPCs for transferring them between clients and servers. NFSv4 ACLs are similar to Microsoft Windows ACLs [14]. Unfortunately, the designers of NFSv4 have mostly ignored the existence of POSIX ACLs, so NFSv4 ACLs are not compatible with POSIX ACLs. Marius Aamodt Eriksen describes a one-way mapping between POSIX ACLs and NFSv4 ACLs [6], but this mapping is impractical. One of the central concepts in POSIX ACLs, which is needed to ensure compatibility with legacy POSIX.1 applications, is the mask entry. The NFSv4 ACL model could be extended by the mask concept. Although this would greatly improve interoperability with POSIX ACLs, proposals to extend the NFSv4 specification have so far been rejected.

Partial NFSv3 support has been added in the 2.2 Linux kernel series. The ACCESS RPC was added to the kernel NFS daemon in version 2.2.18, but the NFS client only correctly uses the ACCESS RPC in the 2.5 kernel series. A patch for older kernels exists since kernel version 2.4.19, which is included in the SuSE and UnitedLinux products.

Because the ACCESS RPC can lead to noticeable network overhead even on file systems that are known not to include any ACLs, the Linux NFSv3 client allows to mount file systems with the *noacl* mount option. Then the NFS client will use neither the ACCESS RPC nor the GETACL or SETACL RPCs. To ensure that no ACLs can be set on the server, the Ext2, Ext3, JFS, and ReiserFS file systems can be mounted on the server without ACL support by omitting the *acl* mount option.

Since March 3, 2003, an implementation of Sun's NFS ACL protocol for Linux (which is also included in SuSE Linux 8.2) is available at <http://acl.bestbits.at/>, with friendly permission from Sun to use it. The NFS ACL protocol was chosen because it is simple and sup-



ports POSIX 1003.1e draft 17 ACLs well enough. Solaris ACLs are based on an earlier draft of POSIX 1003.1e, so its handling of the mask ACL entry is slightly different than in draft 17 for ACLs with only four ACL entries. This is a corner case that occurs only rarely, so the semantic differences may not be noticeable.

## 12 Samba and ACLs

Microsoft Windows supports ACLs on its NTFS file system, and in its Common Internet File System (CIFS) protocol [20], which formerly has been known as the Server Message Block (SMB) protocol. CIFS is used to offer file and print services over a network. Samba is an Open Source implementation of CIFS. It is used to offer UNIX file and print services to Windows users. Samba allows POSIX ACLs to be manipulated from Windows. This feature adds a new quality of interoperability between UNIX and Windows.

The ACL model of Windows differs from the POSIX ACL model in a number of ways, so it is not possible to offer entirely seamless integration. The most significant differences between these two kinds of ACLs are:

- Windows ACLs support over ten different permissions for each entry in an ACL, including things such as append and delete, change permissions, take ownership, and change ownership. Current implementations of POSIX.1 ACLs only support read, write, and execute permissions.
- In the POSIX permission check algorithm, the most significant ACL entry defines the permissions a process is granted, so more detailed permissions are constructed by adding more closely matching ACL entries when needed. In the Windows ACL model, permissions are cumulative, so permissions that would otherwise be granted can only be restricted by DENY ACL entries.
- POSIX ACLs do not support ACL entries that deny permissions. A user can be denied permissions by creating an ACL entry that specifically matches the user.
- Windows ACLs have had an inheritance model that was similar to the POSIX ACL model. Since Windows 2000, Microsoft uses a dynamic inheritance model that allows permissions to propagate down the directory hierarchy when permissions of parent directories are modified. POSIX ACLs are inherited at file create time only.
- In the POSIX ACL model, access and default ACLs are orthogonal concepts. In the Windows ACL model, several different flags in each ACL entry control when and how this entry is inherited by container and non-container objects.
- Windows ACLs have different concepts of how per-

missions are defined for the file owner and owning group. The owning group concept has only been added with Windows 2000. This leads to different results if file ownership changes.

- POSIX ACLs have entries for the owner and the owning group both in the access ACL and in the default ACL. At the time of checking access to an object, these entries are associated with the current owner and the owning group of that object. Windows ACLs support two pseudo groups called Creator Owner and Creator Group that serve a similar purpose for inheritable permissions, but do not allow these pseudo groups for entries that define access. When an object inherits permissions, those abstract entries are converted to entries for a specific user and group.

Despite the semantic mismatch between these two ACL systems, POSIX ACLs are presented in the Windows ACL editor dialog box so that they resemble native Windows ACLs pretty closely. Occasional users are unlikely to realize the differences. Experienced administrators will nevertheless be able to detect a few differences. The mapping between POSIX and Windows ACLs described here is found in this form in the SuSE and the UnitedLinux products, while the official version of Samba has not yet integrated all the improvements recently made:

- The permissions in the POSIX access ACL are mapped to Windows access permissions. The permissions in the POSIX default ACL are mapped to Windows inheritable permissions.
- Minimal POSIX ACLs consist of three ACL entries defining the permissions for the owner, owning group, and others. These entries are required. Windows ACLs may contain any number of entries including zero. If one of the POSIX ACL entries contains no permissions and omitting the entry does not result in a loss of information, the entry is hidden from Windows clients. If a Windows client sets an ACL in which required entries are missing, the permissions of that entry are cleared in the corresponding POSIX ACL.
- The mask entry in POSIX ACLs has no correspondence in Windows ACLs. If permissions in a POSIX ACL are ineffective because they are masked and such an ACL is modified via CIFS, those masked permissions are removed from the ACL.
- Because Windows ACLs only support the Creator Owner and Creator Group pseudo groups for inheritable permissions, owner and owning group entries in a default ACL are mapped to those pseudo groups. For access ACLs, these entries are

mapped to named entries for the current owner and the current owning group (e.g., the POSIX ACL entry “u:rw” of a file owned by Joe is treated as “u:joe:rw”).

If an access ACL contains named ACL entries for the owner or owning group (e.g., if one of Joe’s files also has a “u:joe:...” entry), the permissions defined in such entries are not effective unless file ownership changes, so such named entries are ignored. When an ACL is set by Samba that contains Creator Owner or Creator Group entries, these entries are given precedence over named entries for the current owner and owning group, respectively.

- POSIX access ACL and default ACL entries that define the same permissions are mapped to a Windows ACL entry that is flagged as defining both access and inheritable permissions.

### 13 Backup and Restore

An important but easily overlooked aspect of introducing new features like EAs and ACLs is backup. Standard tools like GNU tar and GNU cpio do not include EA or ACL support. The *ustar* and *cpio* archive formats on which these tools are based do allow certain extensions, but no standards for storing EAs or ACLs have yet been defined.

The current version of POSIX.1 [11] defines a new utility called *pax*, which stands for *portable archive interchange*. The *pax* utility supports both the *ustar* and *cpio* archive formats in addition to the new *pax* archive format. This new format is based on *ustar* and is, to a large degree, compatible with *ustar*. *Pax* introduces a mechanism called *extended headers*. Extended headers consist of a list of attributes very similar to EAs. They are used to store things that cannot be represented in *ustar* headers, such as sub-second resolution file timestamps, or file sizes of 8 GiB or more.

The *pax* archive format is a good match for storing EAs and ACLs. As neither EAs nor ACLs are part of any POSIX standard, no specific format for EAs or ACLs to use in extended headers has been defined. The specification leaves room for vendor-specific attributes tagged with the vendor name, so even if no agreement can be reached soon on the EA or ACL formats to be used, the vendor-specific extensions can at least be distinguished and implementations may support more than one extension.

A benefit of the *pax* format is that most *pax* archives can also be restored with tar implementations. To tar, extended headers look like files of an unknown type. The information stored in the extended headers will be lost, but files and directories will be extracted correctly. This will not work for *pax* archives that contain files 8 GiB or more in size; this is the maximum file size in tar archives.

The following solutions exist for backing up (and later restoring) EAs and ACLs:

- Jörg Schilling’s implementation of *pax* and tar called *star* includes support for POSIX.1e ACLs and a few others. The resulting archives are portable among systems that implement various versions of POSIX ACLs, including FreeBSD, Irix, HP-UX, Compaq/HP Tru64, Linux, Solaris. A patch that implements Linux EA support in *star* exists as well. *Star* is available from <ftp://ftp.berlios.de/pub/star/>.
- On the XFS file system, the *xfsdump* and *xfsrestore* utilities can be used. However, the backup format is file-system-specific, so this approach is not recommended.
- Finally, the *getfattr* and *setfattr* utilities can dump ACLs and EAs to text files, which the *setfattr* and *getfattr* utilities are able to restore. This works reasonably well for restoring complete backups, but it is impractical for restoring individual files.

### 14 Application Support for ACLs

Today, the most basic tools that are needed to operate a system with EAs and ACLs are available: there is EA and ACL support in the basic file utilities (*ls*, *cp*, and *mv*), there are utilities for manipulating EAs and ACLs from the command line, and there are solutions for backing up and restoring a system that uses those features. Still, there are many applications that are lacking support. Although for many of them this is irrelevant, there are some classes of applications for which this leads to problems. This includes file managers, editors, and file system synchronization tools.

File managers usually can copy and move files and allow permissions to be changed. UNIX kernels have no functions for copying files or for moving files across file system boundaries. Therefore, these operations are implemented by reading from the source file and copying the data to the destination file. The kernel has no way of telling which sequences of operations are file copies or moves and which are something else, so it cannot preserve EAs and ACLs automatically. This means that applications must take care of preserving EAs and ACLs themselves as needed.

Front-ends that allow manipulation of permissions usually allow manipulation of the standard POSIX.1 permissions, but none are known yet that allow manipulation of ACLs. It is quite likely that for the foreseeable future, ACL editing support will not become available except with the command line utilities.

Some editors suffer from the file copying problem as well. There are two ways of safely modifying a file. One is to create a copy of the original file and then to modify the original file. The other is to rename the origi-

nal file and then to create a new file that includes the modifications in the original file's place. The latter is equivalent to copying files as far as EAs and ACLs are concerned. The option used also has additional consequences for files that are symlinks and for files with a link count greater than one. *Emacs* and *vi*, the most popular editors on UNIX-like systems, both can be configured to use either method.

When preserving permissions, it is important that as much information is preserved as possible. Although this seems obvious, correctly implementing this is not trivial. There are a number of complications that make that operation prone to implementation errors. This is especially true if the source and destination files reside on different file systems, only one of which has ACL support. To take this additional burden from programmers, the current version of *libacl* includes functions for copying EAs and ACLs between files [8].

It would also be nice to have EA and ACL support in popular utilities like *scp* and *rsync*. Unfortunately, utilities that transfer files between different systems have a much harder time handling incompatibilities. Only some UNIX-like systems support the POSIX.1e ACL library functions and other systems have their own operating system interfaces. Even worse, the semantics of ACLs differs widely among UNIX systems alone, not to speak of non-UNIX systems. The semantics and system interfaces for EAs unfortunately are different among various systems as well.

## 15 Conclusions and Future Work

Integrating support for EAs was an important step that is going to simplify the development of various sorts of applications, including system level security extensions, such as Capabilities and Mandatory Access Control schemes, Hierarchical Storage Management, and many user space solutions.

POSIX.1e ACLs are a consequent extension of the POSIX.1 permission model. They support more fine-grained and complex permission scenarios that are difficult or impossible to implement in the traditional model.

It is unfortunate that neither of these areas has been formally standardized. Already there is a wild mix of implementations with subtle differences and incompatibilities. As more implementations become available, future standardization is becoming more and more unlikely.

As for POSIX ACLs, although they are a substantial improvement, many restrictions remain:

- More find-grained permissions would be useful. For directories, the write permission includes the rights to add and remove files. For files, it allows overwriting of existing contents as well as appending. For directories, the sticky bit helps, but its ap-

plicability is limited. Ext2 and Ext3 support flags like *append* and *immutable* that can be set on a per-file basis. ACL permissions would be per-user or per-group.

- The creator of a file is also the initial file owner. There is no way to restrict the file owner from modifying permissions. It is impossible to implement upload directories securely at the file system level that don't allow to modify existing files, or that don't allow users to create files that other users can read.
- It is not possible to completely delegate administration of a directory to a regular user. It would be necessary to ensure that this locally-privileged user cannot be denied access to files below that directory and that this user can change permissions, and potentially also assign or take ownership of files.

On UNIX-like systems, it is easier to work around problems than on other popular systems, but these workarounds cause complexity and may contain bugs. It may be better to solve some of the existing problems at their root. All extensions must be designed carefully to simplify the integration with existing systems like Windows/CIFS and NFSv4.

The UNIX way of identifying users and groups by numeric IDs is a problem in large networks [18]. Like the whole POSIX.1 permission model, current implementations of POSIX.1e ACLs are based on these unique IDs. Maintaining central user and group databases becomes increasingly difficult with increasing network size. The CIFS and NFSv4 protocols solve this problem differently.

In CIFS, users and groups are identified by globally unique security identifiers (SIDs). Processes have a number of SIDs, which determine their privileges. CIFS ACLs may contain SIDs from different domains.

In NFSv4, users and groups are identified by a string of the form "user@domain". Implementations of NFSv4 are expected to have internal representations for local users, such as unique user or group IDs. ACLs may contain local and non-local user or group identifiers.

Current implementations of POSIX ACLs only support numeric user or group identifiers within the local domain. Allowing non-local identifiers in ACLs seems possible but difficult. A consequent implementation would require substantial changes to the process model. At a minimum, in addition to non-local user and group identifiers in ACL entries, file ownership and group ownership for non-local users and groups would have to be supported.

## 16 Acknowledgments

I would like to thank SuSE for allowing me to continue working on ACLs. Much of what was achieved in recent



months would not have been possible without the developers from Silicon Graphics's XFS for Linux project. Many thanks to Robert Watson and Erez Zadok and to the anonymous Usenix reviewers for their many suggestions that have lead to countless improvements in the text.

## References

- [1] Curtis Anderson: *xFS Attribute Manager Design*. Technical Report, Silicon Graphics, October 1993. [http://oss.sgi.com/projects/xfs/design\\_docs/xfsdocs93\\_pdf/attributes.pdf](http://oss.sgi.com/projects/xfs/design_docs/xfsdocs93_pdf/attributes.pdf)
- [2] Austin Common Standards Revision Group. <http://www.opengroup.org/austin/>
- [3] Steve Best, Dave Kleikamp: *How the Journaled File System handles the on-disk layout*. IBM developerWorks, May 2000. <http://www-124.ibm.com/developerworks/oss/jfs/>
- [4] B. Callaghan, B. Pawlowski, and P. Staubach: *NFS Version 3 Protocol Specification*. Technical Report RFC 1813, Network Working Group, June 1995.
- [5] Andreas Dilger: *[RFC] new design for EA on-disk format*. Mailing list communication, July 10, 2002. <http://acl.bestbits.at/pipermail/acl-devel/2002-July/001077.html>
- [6] Marius Aamodt Eriksen: *Mapping Between NFSv4 and Posix Draft ACLs*. Internet Draft, October 2002. <http://www.citi.umich.edu/u/marius/draft-eriksen-nfsv4-acl-01.txt>
- [7] Andreas Grünbacher: *Known Problems and Bugs in the Linux EA and ACL implementations*. March 20, 2003. <http://acl.bestbits.at/problems.html>
- [8] Andreas Grünbacher: *Preserving ACLs and EAs in editors and file managers*. February 18, 2003. <http://www.suse.de/agruen/ea-acl-copy/> for a description.
- [9] Hewlett-Packard: *acl(2): Set a file's Access Control List (ACL) information*. HP-UX Reference. <http://docs.hp.com/>
- [10] Hewlett-Packard: *acl(4): Access control list*. Compaq Tru64 Reference Pages. <http://www.hp.com/>
- [11] IEEE Std 1003.1-2001 (Open Group Technical Standard, Issue 6), *Standard for Information Technology—Portable Operating System Interface (POSIX) 2001*. ISBN 0-7381-3010-9. <http://www.ieee.org/>
- [12] IEEE 1003.1e and 1003.2c: *Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) and Part 2: Shell and Utilities, draft 17 (withdrawn)*. October 1997. <http://wt.xpilot.org/publications/posix.1e/>
- [13] Jim Mauro: *Controlling permissions with ACLs*. Describes internals of UFS's shadow inode concept. SunWorld Online, June 1998.
- [14] Microsoft Platform SDK: *Access Control Lists*. <http://msdn.microsoft.com/>
- [15] Mark Lowes: *Proftpd: A User's Guide* March 31, 2003. <http://proftpd.linux.co.uk/>
- [16] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, D. Noveck: *NFS version 4 Protocol*. Technical Report RFC 3010, Network Working Group, December 2000.
- [17] Silicon Graphics: *acl(4): Access Control Lists*. Irix manual pages. <http://techpubs.sgi.com/>
- [18] J. Spadavecchia, E. Zadok: *Enhancing NFS Cross-Administrative Domain Access*. Proceedings of the Annual USENIX Technical Conference, FreeNIX Track, Pages 181–194. Monterey, CA, June 2002.
- [19] W. Richard Stevens: *Advanced Programming in the UNIX(R) Environment*. Addison-Wesley, June 1991 (ISBN 0-2015-6317-7).
- [20] Storage Networking Industry Association: *Common Internet File System Technical Reference*. Technical Proposal, March 2002. [http://www.snia.org/tech\\_activities/CIFS/](http://www.snia.org/tech_activities/CIFS/)
- [21] Sun Microsystems: *acl(2): Get or set a file's Access Control List*. Solaris 8 Reference Manual Collection. <http://docs.sun.com/>
- [22] Sun Microsystems: *NFS: Network file system protocol specification*. Technical Report RFC 1094, Network Working Group, March 1989.
- [23] Robert N. M. Watson: *acl(3): Introduction to the POSIX.1e ACL security API*. FreeBSD Library Functions Manual. <http://www.FreeBSD.org/>
- [24] Robert N. M. Watson: *TrustedBSD: Adding Trusted Operating System Features to FreeBSD*. USENIX Technical Conference, Boston, MA, June 28, 2001. <http://www.trustedbsd.org/docs.html>
- [25] Robert N. M. Watson: *Introducing Supporting Infrastructure for Trusted Operating System Support in FreeBSD*. BSDCon 2000, Monterey, CA, September 8, 2000. <http://www.trustedbsd.org/docs.html>
- [26] Robert N. M. Watson: *Personal communication*. March 28, 2003.
- [27] Winfried Trümper: *Summary about Posix.1e*. Publicly available copies of POSIX 1003.1e/1003.2c. February 28, 1999. <http://wt.xpilot.org/publications/posix.1e/>



# Privman: A Library for Partitioning Applications

Douglas Kilpatrick  
*Network Associates Laboratories*

## Abstract

Writing secure, trusted software for UNIX platforms is difficult. There are a number of approaches to enabling more secure development, but it is apparent that the current set of solutions are neither achieving acceptance nor having sufficient impact. In this paper, we introduce a library to address a particularly difficult problem in secure code development: partitioning processes to isolate privileges in trusted code.

Privilege separation is a technique that isolates trusted code, therefore reducing the amount of code that needs to be carefully audited. While the technique is not new, it is not widely used due to difficulties of implementation. We present Privman, a library that makes privilege separation easy. The primary benefit of the Privman library is a systematic, reusable framework and library for developing partitioned applications. We demonstrate the feasibility of the approach by applying it to two real systems, thttpd and WU-FTPD.

## 1. Introduction

According to the SANS list of critical security vulnerabilities [SAN02], simple programming errors such as buffer overflows cause approximately 40% of the top vulnerabilities. Despite a large collection of knowledge on writing secure software, programmers still make the same mistakes today as 10 years ago, and the same types of services cause most of the problems.

Standard UNIX systems do not handily support the concept of least-privilege, instead granting all privileges to superuser (root), and none otherwise. While specific systems may have some least-privilege features [IEE97], the relevant APIs have not achieved sufficiently widespread usage, and therefore cross-platform applications are not able to take advantage of them. Although the widespread use of techniques like those in StackGuard [COW98a] would protect against these common buffer overflows, even the strongest compiler techniques are useless against application design errors.

There are a number of reasons for the lack of progress in developing secure software. The methodologies for developing secure software are not used, as general application developers with little security awareness write most software. Additionally, the existing body of knowledge commonly available to developers consists of a list of missteps, not positive guidelines [KIE02a]. Even when the developers know of the issues, they often drop security concerns in the rush to add functionality.

Finally, the secure operating system features that would allow secure software to be written more easily have not achieved widespread deployment for a number of different reasons [COW98], including deployment costs and more general networking/interoperability issues.

Secure programming practices have significant value, but most developers are not security experts and therefore cannot take full advantage of those practices. This trend will not change. Therefore, we seek to lower the barriers of entry for writing secure system software.

Our contribution towards this goal involves facilitating a specific technique for writing secure software: partitioning applications between security-specific (trusted) code and non-security-specific (untrusted) code. By validating requests in the privileged code, an application can be limited to the required set of privileges.

## 2. Problem: Partitioning Applications

Software is, by its very nature, extremely sensitive to mistakes [BRO75]. Privileged software is no different. In all common operating systems, software with the ability to perform *any* privileged operation also has the ability to perform *all* privileged operations. Therefore, this software must be trusted to not misbehave, even when under attack from untrusted sources. When writing trusted software, a single mistake can compromise the entire system.

An important defensive programming technique is encapsulating the security-sensitive components within small, simple components. These components can then be more easily verified. This pattern [KIE02b] of separation provides additional assurance when requests for privileged operation have to be validated by the secure component. By separating the security sensitive components of the application in a different process from the bug-prone components, mistakes in the majority of the application will not and cannot result in total compromise of the entire application. Hence, attackers would be unable to compromise the system.

The most common example of this pattern of enforced separation is the kernel/user-space split of the major operating systems, but this is also the pattern of any trusted computing base (TCB). This split can be applied to user-space software by splitting the software among processes [VIE98].

On UNIX systems, many exercises of privilege fundamentally exist as file accesses. For example, validating a user's authentication normally requires read access to the `/etc/passwd` and `/etc/shadow` files. UNIX systems validate file-access privilege upon `open(2)` only, and therefore consider file descriptors to be effectively access tokens.

The standard POSIX APIs allows file descriptors to be passed between processes via UNIX domain sockets (pipes)<sup>1</sup>[POS00]. This API therefore allows software to pass privilege tokens between processes. Programs can proxy many useful operations by passing file descriptors between processes.

Several projects have used this POSIX capability to split their process in a one-off and ad-hoc way, including recent versions of the ubiquitous OpenSSH daemon [PRO02]. Several other projects have used other forms of process compartmentalization, including gmail [NEL02] and postfix [BLU01].

Traditionally, it has been very difficult to retrofit this design to existing applications. Instead, almost all such applications have been written from the ground up around this design.

The Privman library seeks to overcome this limitation by allowing developers to easily adapt existing applications. The Privman library provides a systematic and reusable approach to enabling development of a partitioned application.

## 2.1. Constraints

For developers to adopt our approach, it must fit into the "real world" of application development. This places several constraints upon any solution.

- The framework must exist in a form convenient for development. We believe this requires the system to be available as a library.
- The framework must be portable between most UNIX systems.
- The framework cannot rely upon custom kernel changes or custom system libraries.
- The library API should be in terms of existing, well-understood APIs, simplifying any porting efforts.

The barriers to changing a piece of the core infrastructure of a distribution (like the compiler or the fundamental nature of the security policy of the kernel) are high. In contrast, the barriers to small changes in a specific daemon are low. (A minimal install of Red Hat Linux version 7.2 includes approximately 70 packages of libraries, most of which are only used for one or two different pieces of software.) We expect, therefore, that a well-designed library may lure developers of system software.

## 2.2. Related Work

Many systems have approached the problem of least privilege, even under the constraints listed above; OpenSSH, for example, currently uses a framework very similar to Privman to protect systems from possible bugs in the ssh daemon. However, the privilege-separation framework in OpenSSH is application-specific in many ways. The majority of the privileged operations in privilege separated OpenSSH are SSH-specific operations, and the framework assumes that application-specific operations are the norm. Privman does not make that assumption, and is more useful for general applications.

Other projects have modified the kernel on systems to enforce application-specific security policies. Systrace[PRO03], GSWTK[FRA99], and Tron[BER95], mediate at the system call layer to enforce their policy. We avoided this approach to improve portability across systems. SubOS[IOA02] uses a modified kernel to associate permissions with files, and restrict the permissions of programs that open those files. While using a modified kernel is faster and offers higher levels of granularity, prior approaches have not achieved

widespread adoption, although Systrace may break through.

Similar to the kernel-based sandbox systems are the systems that enforce a policy on running applications by using system-specific debugging interfaces. Janus [GOL98] and MAPbox [ACH00] are two examples of this approach. These systems are less portable across platforms. They can also suffer performance issues, as each system call requires the intervention of a user-space application. Privman may be slightly quicker, as only those calls that require privilege need to be passed to the Privman server.

None of these sandbox systems require modification of the running applications. On the other hand, application-specific systems do not require custom kernel builds or non-standard debugging APIs. In all cases, developing a correct and strict policy is the primary difficulty. Systrace, unlike GSTWK, can automatically build policies for applications running under the system, but the policy-building process for Privman and most of sandboxing systems is still fairly difficult and manual.

Various projects such as qmail and vsftpd have developed application-specific methods of privilege separation, normally involving process separation and clean interfaces between processes. These approaches are used only in the creation of new software, as it is impractical to adapt existing software to this design.

Several projects restrict the privileges given to daemons by placing the daemons in some form of “sandbox”. FreeBSD has its `jail(2)` system call, Linux has User-Mode-Linux[UML03], and many systems have some form of total-system virtualization. The Java VM’s security manager may be the most famous sandbox. Privman is more portable across common systems than most of these sandboxing techniques, and unlike the Java VM, supports C and C++.

### 3. Solution: The Privman Library

We present a library, called Privman, which simplifies the task of partitioning applications for a particular class of applications, privileged UNIX daemons.

Programs that use Privman split themselves into two processes: a privilege server, and the main application. The main application gives up all privilege, and asks the privilege server to perform any privileged operations on its behalf.

```
int main(int argc, char *argv[])
{
    char buf[4096];
    int i;
    priv_init("mycat");

    for (i=1; i < argc; ++i) {
        /* Privileged use of "fopen" */
        FILE *f=priv_fopen(argv[i], "r");
        while(n=fread(buf, 1, 1024, f) > 0)
            fwrite(buf, n, 1, stdout);
        fclose(f);
    }
    exit(0);
}
```

**Listing 1**

```
struct pam_conv conv = {
    misc_conv,
    NULL
};

int main(int argc, char *argv[])
{
    pam_handle_t *pamh = NULL;
    const char *user = argv[1]
    priv_init("check_user");
    if (priv_pam_start("login", user,
        &conv, &pamh) != PAM_SUCCESS)
        goto failed;
    if (priv_pam_authenticate(pamh, 0)
        != PAM_SUCCESS)
        goto failed;
    if (priv_pam_acct_mgmt(pamh, 0)
        != PAM_SUCCESS)
        goto failed;
    fprintf("user %s authenticated "
        "%s\n", user);
    return EXIT_SUCCESS;
failed:
    fprintf("could not authenticate %s\n",
        user);
    return EXIT_FAILURE;
}
```

**Listing 2**

Privman uses an application-specific configuration file to limit the available privileges. The policy can limit a program to opening specific files, binding to specific ports, or otherwise limit access to the privileged operations.

The Privman library makes implementing privilege separation much easier, by providing standard C library equivalent functions for many operations that

traditionally need privilege. The library supports several file-access methods, PAM authentication, `bind()`, and `daemon()`. The library supports several "`<foo>_as()`" methods to enable changes in the effective unprivileged user.

By limiting the amount of active code that runs with privilege, the amount of code requiring serious audit also shrinks. This should improve overall system security by increasing the assurance of the system.

### 3.1. Usage of the Library

The library has a very straightforward API at its core. The process starts by calling "`priv_init()`", and then calls various "`priv_*`" methods when it wants to perform privileged operations. For example, a variant of `cat` that uses the library to read otherwise unreadable files might look like listing 1 (Privman specific parts in bold). Similarly, a stripped down program that authenticated a user is shown in listing 2.

As shown here, not all requests handled by the library are file-related. The Privman libraries can manage any request that can be proxied (in the case of PAM, by invoking input functions in the context of the unprivileged process).

The Privman library handles two types of extensions. Info functions return a string, and capability functions return a file descriptor. Both types take an array of `char *`. To register an extension function, the program calls `priv_register_{cap,info}_fn()` before it calls `priv_init()`. (Allowing the process to add extension methods after `priv_init()` is essentially allowing the process to run arbitrary code as root at any time.) The register functions return a handle which is used as an argument to later calls to the invoke functions. If a program needs to transition to a different user, it uses any of the "`_as()`" methods. All the "`as`" methods take a user name and a chroot jail, then spawn a second process to continue execution as the specified user.

The complete list of supported APIs is presented in Appendix 1.

### 3.2. Policy

Every Privman managed application has a config file, which in the case of a logfile review program might look like:

```
open_ro {
    # Anything under /var/log
    /var/log/*
}
```

or in the "check\_user" case might look like:

```
# simple app. Only needs PAM.
auth true
```

The proper configuration file is critical to successfully partitioning a process. Obviously, if the privilege manager automatically responds to any request, then Privman would only provide an illusion of security. The attacker would simply rewrite shell code to invoke the privilege manager instead of directly attacking the system. Instead, Privman relies on the configuration file to specify tight constraints on the allowable actions of a client.

This has the extra benefit of expressing the security policy openly, instead of leaving it buried in the code.

The configuration file should be written tightly enough to allow the process its privileged operations but nothing else, *i.e.*, the policy should follow the least privilege principle. For example, the following is the configuration for a simple network daemon.

```
# echo daemon. The app is allowed
# to bind to a low port: 7
# and to write to a log file
bind echo
open_aio {
    /var/log/myecho.log
}
fork true
```

The grammar for the policy file is presented in Appendix 2.

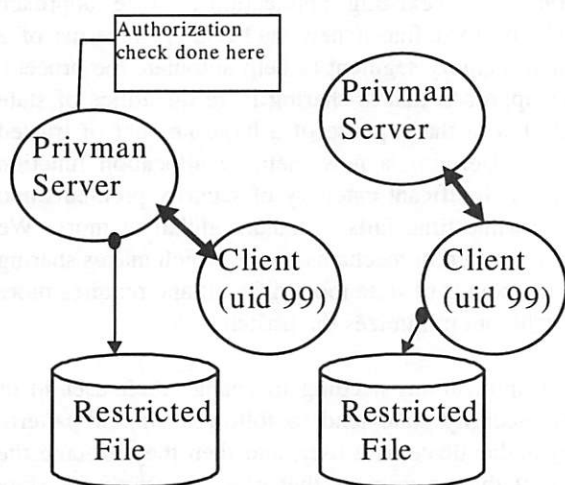
### 3.3. Implementation

The Privman server performs all the privileged operations. When the application invokes a privileged operation, the library marshals the arguments, sends them over a UNIX pipe to the Privman server, and the operation is invoked there. Consequently, operations that do not actually need privilege may need to be executed by the Privman server. As an example, only `pam_authenticate()` really needs privilege to execute, but the Privman library proxies all of the PAM methods so that the state in the libpam library is consistent.



The Privman server is created from the main process during the call to `priv_init()`. The process, when it returns from `priv_init()`, is actually a child process forked off from the Privman server.

This design causes the library to export more methods than might be expected. For example, the `daemon()` call, which detaches a program from the controlling terminal, does not need privilege. However, any shell waiting on the daemon to exit will be waiting on the Privman server. The Privman server, then, needs to call `daemon()` to fully detach from the controlling terminal.



**Figure 1**

Figure 1 represents the program in listing 1 after it opens the file and starts to read. The Privman server passes the file descriptor to the Privman client, and the client is now reading freely, without involving the server for read calls.

The Privman server constitutes of very little code, only approximately 1400 lines (including comments) in the Privman 0.9.1 release. The equivalent code from the application-specific OpenSSH privilege separation framework is approximately 1500 lines<sup>ii</sup>. By limiting the amount of code running with privilege, we also limit the amount of code that requires serious auditing.

## 4. Security Properties

Assuming we code the Privman libraries correctly, programs that use Privman are incapable of performing privileged operations not allowed by the policy, even if the program's logic is compromised by an attacker. However, if the policy for an application is

sufficiently permissive, attackers may be able to achieve their goals inside the constraints of the policy.

Generating a correct and minimal policy is difficult. Automatically generated policies, like in Systrace, can provide a quick first estimate of least privilege, but do not attempt to determine if an application is using privilege it does not actually need.

We have made no attempt to formally prove the correctness of Privman. The small size of the Privman server allows for careful auditing of the privileged code. Additionally, Privman is fail-closed. Whenever Privman detects error conditions, it makes no attempt to recover. Instead, the privilege server exits after logging the error message to the system's logs. Without a privilege server, the application is unable to perform any further privileged operations.<sup>iii</sup>

The largest area of vulnerability involves the protocol for communication between the privilege server and the client. This code runs with privilege and also processes incoming requests; the code is therefore vulnerable to malicious input. Fortunately, the amount of code required to process the incoming request is extremely small.

Additionally, the policy language itself should be carefully analyzed to make sure that policies can express least privilege.

## 5. Open Issues

The majority of issues stem from the decision to run the unprivileged portions of the program in a child process. By changing around the processes, we break reasonable assumptions of state inheritance in `fork()`. Some assumptions about the process identifier (pid) of the active process are also broken.

### 5.1. API Design Issues

The work, as originally proposed, involved a single Privman server per system<sup>iv</sup>. The single server would be contacted by all managed clients, and it would perform privileged operations on behalf of the entire system.

This design would allow programs to start execution without having any real pre-existing privilege outside of an ability to authenticate themselves with the privilege manager. Unfortunately, on stock UNIX systems,

there is no practical way to verify the identity of a process<sup>5</sup>.

Without that verification, this design would invite identity attacks, where an attacker would attempt to spoof the identity of a permitted privileged process. Rather than try to solve this authentication problem, we decided to leverage the pre-existing source of privilege: the fact that a process is already running in a privileged state.

The Privman managed process starts `main()` with heightened traditional (root) permissions. The `priv_init()` call then divides the process into two separate processes: an unprivileged child that runs the original program, and a parent that becomes the Privman server.

From the perspective of the original process, after the `priv_init()` call the process is no longer running as a privileged process and is instead running as the “nobody” user<sup>6</sup>. Any privileged operations must be proxied and validated by the Privman server.

By splitting a process in this manner, we simplified the design of the Privman server, decreasing the probability of serious coding flaws.

Our approach has significant drawbacks when compared to traditional Mandatory Access Control-style secure systems. For example, we are unable to handle any concept of revocation. We are unable to handle permissions at granularity smaller than file access (or other mediated call). In addition, not all security decisions on a UNIX system are in terms of files or file descriptors.

In particular, there are a small number of calls that change the security context of the current process. One obvious example is `chroot()`, but the list also includes `limit()` and the very important `setuid()` family. Much of the software in question may need to make these changes to its security context, but our design makes accommodating this software difficult.

Obviously, other projects that use this type of process split have run into similar issues. We chose to model our approach on the OpenSSH solution [PRO02].

In the OpenSSH solution, the client process packages up state and sends it back to the parent. The parent then creates a second child process with the desired security context and state, and allows that child to continue execution in that context.

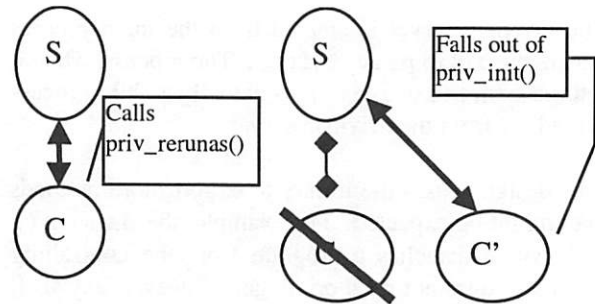


Figure 2

The problems of packaging up and managing state are considerable when attempting to retrofit the Privman library to preexisting applications. One approach might be to define a new `malloc()` in terms of a shared memory segment to help automate the process. This approach makes sharing large quantities of state easy, but at the expense of a large amount of trusted code. Defining a new memory allocation function opens a significant category of security problems and at the same time fails to handle global memory. We chose a different mechanism, one which makes sharing large amounts of state more difficult and requires more thought, but minimizes the trusted code.

Most applications needing to change their user id or other security state tend to follow a similar pattern: they authenticate as a user, and then they execute the bulk of the program as that user. Most of the state built up during the authentication stage is incidental, and can be safely ignored. Therefore, the amount of state that needs to be moved between processes is minimized.

Our library provides a “`rerunas()`” method. This method causes the Privman server process to re-execute “`priv_init()`”. The Privman server process forks a second child, and the second child becomes the specified user. It then returns from `priv_init()` with the program’s state being basically the same as when `priv_init()` was originally called. To distinguish between the original case, and the `rerunas` case, the calling process can supply a function pointer and an arguments string vector. The function will be invoked in the context of the new child, allowing the child to set up global state so that the proper code paths will be followed. The calling process will exit, transferring control of the application logic to the new child process.

This method allowed the port of `wu-ftpd` to support non-anonymous users easily. Tracking all the state that changed during execution is equivalent to process

migration. Instead, we simply changed `wu-ftpd` so that it could run as if the user had already been authenticated. The structural changes to `wu-ftpd` were minimal.

## 5.2. Security weaknesses

One of the advantages to performing process partitioning manually is a heightened ability to match the policy to the application's specific needs. For example, the OpenSSH daemon knows that certain privileged requests can only happen once and so only allows them once.

The Privman library is unable to know the security requirements of an application to the same degree. The configuration, while more detailed than traditional Unix system privilege, may allow an overly broad amount of privilege to an attacker. As an example, the policy of the `_as()` methods has changed several times as we developed the library.

Programs that require more finely-tuned security policies than are provided by the library's policy are encouraged to write their privileged methods using the extension framework. Applications that use the extension framework have the ability to perform arbitrary permission checks before performing any privileged operations.

## 6. Feasibility

For a best-case example of conversion, we converted the `thttpd` server to use the Privman libraries. `thttpd` is a simple, small, portable, fast and secure HTTP server. The version we used, 2.20c, consists of approximately 2800 lines of C code. The conversion process changed 26 of them, or less than 1% of the code base. The patch is available for download from the Privman web site, <http://opensource.nailabs.com/privman/>.

`thttpd` is essentially a best-case scenario. The server does no complex credentials management, and the preexisting user id management mapped well into the Privman usage pattern. Only four library calls needed to be supported: `open()`, `fopen()`, `bind()`, and `daemon()`. The Privman version actually gets simpler in some respects, as the user id management can be handled by Privman and moved out of the daemon.

In contrast, converting a standard BSD ftp server is closer to a worst-case scenario. The standard FTP server does significant work before it changes the user

id of the server. Consequently, the process has built up large state that must be managed. To use `priv_rerunas()`, some reworking of the base code is required.

`Wu-ftpd`, a derivative of the standard BSD ftp server, totals approximately 32,000 lines of C code, ignoring the build system. The patch only changes 75 lines of C code, most of which are simple replacements of privileged calls with `priv_<foo>` calls. The remainder of the patch merely changes the build system to detect and build against the Privman library.

We believe these examples demonstrate that porting server processes to the Privman library is simple and non-invasive. While there are applications Privman does not handle well in its current state, most server applications should be easily handled.

Privilege separation has shown itself a valuable tool in hardening software against coding errors. OpenSSH has already had one vulnerability stopped by turning on privilege separation, and constructing a demonstration of the technique is as easy as it is useless. Only active deployment by commonly used servers will show the long-term value of the technique.

## 7. Performance

Privilege separation techniques have performance implications, given the two-process model and the IPC involved. To evaluate these costs, we conducted both micro- and macro-benchmarks. The micro-benchmarks measure the performance costs of specific operations, the macro-benchmark measures the effect on a larger system.

For a macro benchmark, we used the `dkftpbench-0.45` ftp benchmark. This benchmark measures the maximum number of concurrent low-bandwidth users an ftp server can support while maintaining a minimum level of responsiveness. We tested the system with both stock `wu-ftpd` and the Privman-enabled `wu-ftpd`. The two versions of the ftp server were compiled from the exact same code, differing only in options presented to the configure script. The system was not optimized for FTP, as the goal was to isolate changes from the introduction of Privman, not maximizing FTP server speed.

The server system was a P3-866 Dell Optiplex GX110. The client loads were produced by a P3-1G Dell Latitude, connected to the server by an otherwise empty



100Mb Ethernet link. The benchmark repeatedly downloaded a ten-kilobyte file. The non-Privman-enabled server supported an average of  $229 \pm 20$  clients over 5 runs, while the Privman-enabled server supported an average of  $217 \pm 4$  users. The difference in performance is approximately 5%. Prior work retrofitting mandatory access controls to common Unix systems have shown similar minimal performance penalties[FRA99].

For a micro-benchmark, we compared the `priv_*` operation to the `libc` equivalent. The benchmark is distributed with Privman as the “microb” test. Slower operations like “`rerunas`” or “`pam_authenticate`” run 10,000 times per execution of the benchmark, and faster operations like “`open`” run 100,000 times per execution of the benchmark.

We ran the benchmark on the server system from the macro benchmark. We ran the benchmark five times, as a memory leak in `pam_authenticate` prevented us from increasing the iterations-per-run. We collected  $\Sigma(x)$  and  $\Sigma(x^2)$  from the runs, and determined the collective average and standard deviations.

When an application invokes any privileged operation, there is a constant overhead of approximately 30 microseconds from the two context switches. Since this overhead is a constant, the simplest operations show the largest performance penalty.

The time to marshal and un-marshal the arguments is also significant. The arguments to `priv_open` and `priv_fopen` are passed as strings, and so take longer to marshal than the arguments to `priv_bind`. In our benchmarks, `priv_open` showed the largest penalty, running 19.6 times slower on average than an equivalent `open(2)`.

The PAM operations took almost exactly the same amount of time whether they were proxied or ran directly. Loading and linking the shared libraries that make up the PAM framework is an expensive operation. This cost may have hidden the costs of marshaling the arguments.

Our experience leads us to believe that the performance penalties for additional access control are generally acceptable in comparison to the security benefits. As a general statement, the operations that require the most careful analysis are also the operations that tend to be slow initially. The performance penalty to an application is therefore minimized, although any particular operation may be a tenth as fast.

Operation	Average	Speed Penalty	Std Dev
<code>bind</code>	8 ms	3.77	13.16%
<code>priv_bind</code>	31 ms		8.01%
<code>open</code>	3 ms	19.6	20.60%
<code>priv_open</code>	64 ms		6.54%
<code>fopen</code>	5 ms	14.1	14.74%
<code>priv_fopen</code>	78 ms		3.46%
<code>pam_auth</code>	4,270 ms	1.06	0.95%
<code>priv_pam</code>	4,540 ms		0.77%
<code>fork+exit</code>	9,780 ms	2.15	1.08%
<code>rerun</code>	21,000 ms		0.56%

## 8. Conclusion

We have presented an overview of the Privman library. The Privman library enables applications to easily take advantage of process partitioning, which will help them become robust against certain types of programming errors.

Applications that use the Privman library can express the required security policy at a fine level of granularity. Consequently, Privman managed applications can approximate least-privilege in a way not common on UNIX systems.

Certain types of applications, primarily simple UNIX servers, are trivial to convert to using the Privman library. Other classes of applications, mainly those that need to change their security policy during normal execution, are less easily handled, but code changes are minimal.

For most cases, applications that use the Privman library will continue to perform adequately. The improvements to security justify any measurable performance degradation, although applications should be careful not to invoke Privman methods in the middle of tight time-sensitive loops.

The current release of the Privman libraries can be found at <http://opensource.nailabs.com>.

## 9. Bibliography

- [BER95] Berman, Andrew, Virgil Bourassa, Erik Selberg, “TRON: Process-Specific File Protection for the UNIX Operating System”, USENIX 1995.



- [ACH00] Acharya, Anurag, Mandar Raje, "MAPbox: Using Parameterized Behavior Classes to Confine Applications", USENIX Security, August 2000
- [BLU01] Blum, Richard. *Postfix*. Sams Publishing, 2001.
- [BRO75] Brooks, Frederick. *The Mythical Man Month*. Addison-Wesley, 1975.
- [COW98a] Cowan, Crispin, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. "Stack Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". USENIX Security, January 1998.
- [COW98b] Cowan, Crispin, Calton Pu, and Heather Hinton. "Death, Taxes, and Imperfect Software: Surviving the Inevitable". New Security Paradigms Workshop, September 1998.
- [COW00] Cowan, Crispin, Heather Hinton, Calton Pu, and Jonathan Walpole. "The Cracker Patch Choice, An Analysis of Post Hoc Security Techniques". NISSC, October 2000.
- [FRA99] Fraser, Timothy, Lee Badger, Mark Feldman, "Hardening COTS Software with Generic Software Wrappers". IEEE Symposium on Security and Privacy, May 1999.
- [GOL98] Goldberg, Ian, David Wagner, Randi Thomas, Eric A. Brewer, "A Secure Environment for Untrusted Helper Applications". USENIX Security, 1998.
- [IEE97] Portable Operating System Interface (POSIX) – part1: System Application Program Interface (API): Protection, Audit and Control Interfaces. C Language, PSSG/DB Posix.1e October 1997.
- [IOA02] Ioannidis, Sotiris, Steven M. Bellovin, Jonathan M. Smith, "Sub-Operating Systems: A New Approach to Application Security." SIGOPS European Workshop, 2002.
- [KIE02a] Kienzle, Darrell, and Matthew Elder. "Security Patterns for Web Development". <http://patterns.nailabs.com>, June 2002.
- [KIE02b] Kienzle, Darrell, Matthew Elder, David Tyree, Jim Edwards-Hewitt. "Partitioned Application". Security Patterns Repository Version 1.0, <http://patterns.nailabs.com>, June 2002.
- [NEL02] Nelson, Russell. "The qmail home page". <http://www.qmail.org/top.html>, June 2002.
- [POS00] Single Unix V3, POSIX 1.1g, from IEEE Std. 1003.1g-2000 draft standard, and RFC2292. [http://www.unix.org/single\\_unix\\_specification](http://www.unix.org/single_unix_specification)
- [PRO02] Provos, Niels. "Privilege Separated OpenSSH". <http://www.citi.umich.edu/u/provos/ssh/privsep.html>, 2002.
- [PRO03] Provos, Niels. "Systrace – Interactive Policy Generation for System Calls". <http://www.citi.umich.edu/u/provos/systrace/>, 2003.
- [SAN02] System Administration, Networking and Security (SANS) Institute. "The Twenty Most Critical Internet Security Vulnerabilities (Updates)". as of May 2, 2002, <http://www.sans.org/top20.htm>, May 2002..
- [UML03] var. "The User-mode Linux Kernel Home Page" as of Feb 28, 2003, <http://user-mode-linux.sourceforge.net/>
- [VIE02] Viega, John, and Gary McGraw. *Building Secure Software*. Addison-Wesley, 2002.

<sup>i</sup> As an object of type SCM\_RIGHTS passed as part of a msg\_control structure via the sendmsg () API.

<sup>ii</sup> This count includes more code from OpenSSH for very application-specific policy, but excludes the actual server implementations of the OpenSSH specific methods.

<sup>iii</sup> The client program may still have access to privileged resources, e.g. if the application opened a protected file, but has not yet closed it.

<sup>iv</sup> Here "system" refers to an actual physical machine.

<sup>v</sup> That is, verify that a given process was actually spawned by a specific binary whose identity can be verified.

<sup>vi</sup> This is, of course, configurable by the program's configuration file.

## Appendix 1: libprivman API

`Priv_init` initializes the Privman server. Call this method first. When this method returns, the application will no longer have root privilege.

```
void    priv_init(const char *appname)
```

These calls act just like the methods they are named after, except that they use the Privman Server. Please see the system man pages for more details about the format of the arguments. The `priv_fork()` method causes both processes to fork, so the child process will still have access to a Privman server.

```
int      priv_open(const char *pathname, int flags, ...);
FILE*    priv_fopen(const char *pathname, const char *mode);
int      priv_unlink(const char *pathname);
int      priv_bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
pid_t    priv_fork(void);
```

These methods use the Privman Server. If the application uses PAM at all, you need to use the `priv_pam_` methods for all PAM methods. Please see the PAM man pages for more details about the format of the arguments.

```
int      priv_pam_start(const char *service, const char *user,
                        const struct pam_conv *conv,
                        pam_handle_t **pamh_p);
int      priv_pam_authenticate(pam_handle_t *pamh, int flags);
int      priv_pam_acct_mgmt(pam_handle_t *pamh, int flags);
int      priv_pam_end(pam_handle_t *pamh, int flags);
int      priv_pam_setcred(pam_handle_t *pamh, int flags);
int      priv_pam_open_session(pam_handle_t *pamh, int flags);
int      priv_pam_close_session(pam_handle_t *pamh, int flags);
int      priv_pam_chauthtok(pam_handle_t *pamh, int flags);
int      priv_pam_set_item(pam_handle_t *pamh, int item_type, const void *item);
int      priv_pam_get_item(pam_handle_t *pamh, int item_type,
                        const void **item);
int      priv_pam_putenv(pam_handle_t *pamh, const char *name_value);
int      priv_pam_getenv(pam_handle_t *pamh, const char *name);
int      priv_pam_fail_delay(pam_handle_t *pamh, unsigned int usec);
```

These methods allow the application to control the process behavior of the Privman server. `priv_exit()` causes the Privman server to exit (and thus prevents you from performing future privileged operations), `priv_daemon()` causes the Privman server to detach from the controlling terminal.

```
void     priv_exit(int status); /* Causes the Privman monitor to exit */
pid_t    priv_wait4(pid_t pid, int *status, int options, struct rusage *rusage);
int      priv_daemon(int nochdir, int noclose);
```

These methods allow the application to change its execution. `priv_execve()` executes the specified program as the user specified and chrooted into the path specified. `priv_popen_as()` works like `popen`, except that the program will run as the user specified. Use `priv_pclose()` to close the stream from `priv_popen_as()`.

```
int      priv_execve(const char *program, char * const argv[],
                    char * const envp[], const char * user,
                    const char* chroot);
FILE     *priv_popen_as(const char *command, const char *type, const char *user);
int      priv_pclose(FILE *stream);
```

This method allows a program to restart, running in a different context. The Privman server will start a new process, running as the user specified and chrooted into the path specified. The new process will execute the function `fnptr` with the supplied arguments, and then will return from `priv_init()`. Both processes will be able to talk to a Privman server.

```
int      priv_respawn_as(void (*fnptr)(char * const *), char * const arg[],
                        const char *user, const char *chroot);
```

This method is like `priv_respawn_as()`, except that the Privman server does not duplicate itself. Pass in `PRIV_RR_OLD_SLAVE_MONITORED` as a flag if the application wants the original process to be able to talk to the Privman server, or 0 if it wants the new process to be able to.

```
int      priv_rerunas(void (*fnptr)(char * const *), char * const arg[],
                    const char *user, const char *chroot, int flags);
```

These methods interface with the extension framework. The application must call the register methods before it calls `priv_init()`, and must call the invoke methods after. The register methods return a handle, which is then passed into the invoke methods to specify which registered method is being invoked.

```
int      priv_register_info_fn(char *(*fnptr)(char * const *));
int      priv_register_cap_fn (int (*fnptr)(char * const *));
char     *priv_invoke_info_fn(int handle, char * const args[]);
int      priv_invoke_cap_fn (int handle, char * const args[]);
```

## Appendix 2: Configuration file grammar

```
config          ⇒    config_stmt_list

config_stmt     ⇒    bind_stmt | open_ro_stmt | open_rw_stmt
                    |    open_ao_stmt | unlink_stmt
                    |    auth_stmt | fork_stmt | rerunas_stmt
                    |    auth_allow_rerunas_stmt | runas_stmt
                    |    unpriv_user_stmt | chroot_jail_stmt

config_stmt_list ⇒    config_stmt_list config_stmt
                    |    config_stmt

% The list of ports the client is allowed to bind to
bind_stmt       ⇒    bind { portlist }

% The lists of files the client is allowed to manipulate
open_ro_stmt    ⇒    open_ro { pathlist }
open_rw_stmt    ⇒    open_rw { pathlist }
open_ao_stmt    ⇒    open_ao { pathlist }
unlink_stmt     ⇒    unlink { pathlist }

% Is the client allowed to use the authentication functions?
auth_stmt       ⇒    auth bool

% Is the client allowed to cause the Privman server to fork?
fork_stmt       ⇒    fork bool

% Is the client allowed to re-execute as a different user?
rerunas_stmt    ⇒    allow_rerun boolean

% If true, any successfully authenticated user is a valid rerunas user.
auth_allow_rerunas_stmt ⇒    auth_allow_rerun boolean

% The client can re execute as any of these users
runas_stmt      ⇒    runas { userlist }

% The default unprivileged user, and starting chroot path.
unpriv_user_stmt ⇒    unpriv_user userid
chroot_jail_stmt ⇒    chroot path

pathlist        ⇒    path pathlist
                    |    ε

portlist        ⇒    port portlist
                    |    ε

userlist        ⇒    user userlist
                    |    ε

user            ⇒    userid
                    |    '*'
```



# The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0

Robert Watson

*Network Associates Laboratories  
Rockville, MD*

[rwatson@nailabs.com](mailto:rwatson@nailabs.com)

<http://www.nailabs.com/>

Chris Vance

*Network Associates Laboratories  
Rockville, MD*

[cvance@nailabs.com](mailto:cvance@nailabs.com)

<http://www.nailabs.com/>

Wayne Morrison

*Network Associates Laboratories  
Rockville, MD*

[tewok@tislabs.com](mailto:tewok@tislabs.com)

Brian Feldman

*FreeBSD Project  
Rockville, MD*

[green@FreeBSD.org](mailto:green@FreeBSD.org)

## Abstract

We explore the requirements, design, and implementation of the TrustedBSD MAC Framework. The TrustedBSD MAC Framework, integrated into FreeBSD 5.0, provides a flexible framework for kernel access control extension, permitting extensions to be introduced more easily, and avoiding the need for direct modification of distributed kernel sources. We also consider the performance impact of the Framework on the FreeBSD 5.0 kernel in several test environments.

## 1 Introduction

Access control extensions have proved a fertile field for operating system security research over the past twenty years: a variety of methods have been employed to extend the system access control policy at great cost to the developers, maintainers, and users of the extended systems. Most of approaches to security extension fall short two vital areas: lack of support by the operating system vendor for various providers of security extensions on the system, and the highly redundant implementation of support infrastructure for security extension providers.

The TrustedBSD MAC Framework included in FreeBSD 5.0 provides a general facility for extending the kernel access control policy [8][17]. By providing common security infrastructure services, such as kernel object labeling, and the ability to instrument kernel access control decisions, the Framework is capable of supporting a variety of policies implemented by different vendors. This paper explores the design, implementation and performance of the MAC Framework, as well as its impact on policy design and the FreeBSD kernel architecture.

## 2 The Desire For Access Control Extensions

FreeBSD serves two primary markets: it is both a consumer operating system and a technology source for third party operating systems or high-end embedded products. FreeBSD is directly employed as a production server and workstation operating system on mainstream i386, Alpha, and SPARC64 hardware. In the high-end embedded market, it is used as the basis for network and storage appliance devices such as firewalls, network-attached storage, and VPN devices; it is also used as a technology source for third party operating system, including Apple's Mac OS X Darwin kernel, as well as by other operating system vendors.

In these three roles, FreeBSD is deployed in a wide variety of environments, ranging from electronic cheque processing and point of sale devices to web cluster deployment, firewall, and routing appliances. Each of these environments has different security requirements, often requiring flexibility beyond that provided for by the traditional UNIX security protections. Especially in embedded network environments, the requirements can range from simple operating system hardening to the introduction of mandatory and fine-grained security policies.

## 3 Access Control Extension Mechanisms

Security research and development literature is rife with approaches to achieving operating system access control extension with (and without) the help of the operating system vendor. Traditional trusted variants of commercial UNIX operating systems have been written by the vendor in response to the needs of specific consumers (such as US DoD) [12][5][9][10]. Typical practice has been to maintain a distinction between the base OS prod-

uct and the trusted variant in terms of maintenance, product identification, and price. In addition, there are third party vendors who develop and market trusted operating system extensions, often in close coordination with the OS vendor[2]. Finally, there is a broad range of access control research across many operating systems and performed in many forms [7] [13] [15]—most frequently, this work is performed on open source operating systems due to ready access to operating system source code.

In order to successfully maintain a security extension product for an operating system, access to the operating system code is typically required, be it an open source system, or licensed from the closed source vendor. Product maintenance raises a number of challenges, not least the challenge of tracking the operating system vendor's primary product life cycle, which is frequently incompatible with the development cycle required for high assurance products. Many practical impediments also present themselves: security extensions have their fingers deep in the heart of the operating system, touching almost all elements of the kernel source code. Local security extensions invariably conflict with vendor-provided security patches, as well as vendor-provided feature improvements over the OS development cycle. In addition, security extensions frequently conflict with one another if deployed in parallel, leading not only to potentially inconsistent policy behavior, but also possible bypass of protections provided by one of the policies. Direct source code modification of the vendor operating system presents many challenges to security extension authors.

In the past, research has been performed on how to most easily extend operating system security policies, including into system-call interposition technologies such as LOMAC [6], Generic Software Wrappers [7], and systrace [?], extensible security mechanisms such as the General Framework for Access Control [1] and FLASK [15]. Many of these extension technologies, especially these using system call wrapping techniques, fall down in the face of modern UNIX operating system kernels which support true kernel and user process parallelism in SMP environments, and fine-grained threading of user processes. Preventing races inherent to system call wrapping is difficult, and most system call wrapper security technologies are susceptible to at least one of a class of related vulnerabilities. Any successful extension technology for contemporary operating systems must be designed with the notion that SMP and threading are realities, and must be well-integrated into the kernel locking mechanisms.

## 4 Motivations for MAC

Mandatory Access Control (MAC) describes a broad class of access control policies; in this context “mandat-

tory” refers to the mandatory imposition of the policy on non-administrative users. Popular mandatory policies including Multi-Level Security (MLS), which enforces mandatory protections based on administrator-defined confidentiality labels, Biba integrity, which enforces system and user data integrity properties, and Type Enforcement, which permits the administrator to define subject domains, object types, and use a policy language to control accesses to objects and other system properties.

Trusted operating systems typically provide two or more mandatory system policies: almost all provide MLS for user data protection, but many also make use of the Biba policy to protect the integrity of the Trusted Code Base (TCB). The TrustedBSD Project, in seeking to provide access to trusted operating system features, provides several MAC policies for use with FreeBSD; the challenges associated with this work include introducing the services securely, and without substantially impacting the performance and reliability of FreeBSD installations not taking advantage of these new features.

## 5 Framework Design and Implementation

The TrustedBSD MAC Framework permits access control policy modules to be loaded into the FreeBSD kernel, providing a tightly integrated security extension vehicle. In order to address the problems identified in Section 3 there are several high-level goals for the design:

- Permit dynamic extension of the kernel access control policy.
- Isolate the logic of access control policies from the implementation of kernel services, permitting the implementations of services to be more mobile in the face of extensions, and reducing OS life cycle issues for policy developers.
- Permit multiple policies to be loaded simultaneously with some useful notion of composition.
- Reduce the redundant infrastructure implementation efforts of policy writers by providing support for common policy infrastructure requirements.
- Integrate tightly with the kernel locking and threading mechanisms to provide correctness and high performance in modern kernel designs.

### 5.1 High Level Design

The MAC Framework is made up of a number of kernel and user-space elements. In the kernel, existing kernel services are modified to add data structure extensions and entry points to the MAC Framework, centralized management of label storage, registration and management of security modules, a series of system calls and sysctls to permit applications to interact with labels and manage the framework, and a series of access policy modules that may be compiled into the kernel or

loaded via loadable kernel modules. In user-space, several new C library interfaces provide access to centralized label configuration via `mac.conf`, and changes to the `libutil` user class and security context management code. Command line tools permit user manipulation of file and process labels, and modifications to standard administrative tools manage system labels.

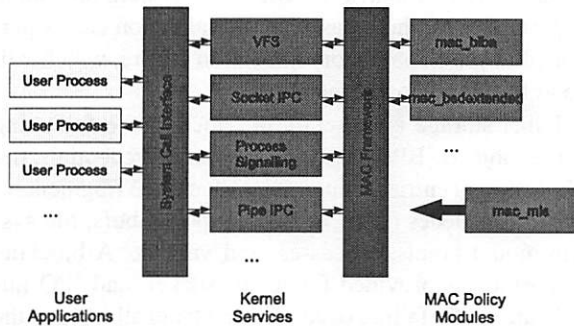


Figure 1: High Level Kernel Design

The MAC Framework addresses a number of needs in access control and extension implementation:

- Policies are encapsulated in kernel modules, which may be linked into the kernel, loaded as part of the boot process, or loaded at run-time in response to environmental requirements.
- Policies are permitted to augment kernel access control decision; sufficient locks to access important elements of check arguments, such as object references, are guaranteed to be held.
- The MAC Framework provides a policy-agnostic labeling service permitting policies to maintain additional meta-data on a variety of system objects. Well-defined locking semantics are provided for object labels, and existing locks on kernel objects typically also protect any labels in the object, permitting atomic checks of both labels and existing object properties without additional locking overhead.
- Policies may back labels into persistent extended attributes provided by UFS and UFS2, permitting labels on file system objects to be maintained while they are not in the in-memory working set.
- When multiple policies are loaded, their access control decisions are usefully composed, where the definition of “useful” is that results are well-defined, may be reasoned about, and are desirable in the context of a number of relevant policies.
- Policies may make use of a policy-agnostic label management API to export access to label data to user processes, as well as permit the management of those labels.

## 5.2 Kernel Services and Objects

The MAC Framework enforces policy over a variety of kernel subsystems and objects, including system configuration interfaces, processes, the file system, IPC primitives, and the network stack. In general, two classes of modifications were made to existing kernel service providers. First, services are modified to invoke MAC Framework entry points during object management, over the course of object life cycles, and when important access control events occur. Second, a number of kernel data structures representing security-relevant objects were modified to include a label structure intended to hold extensible security information.

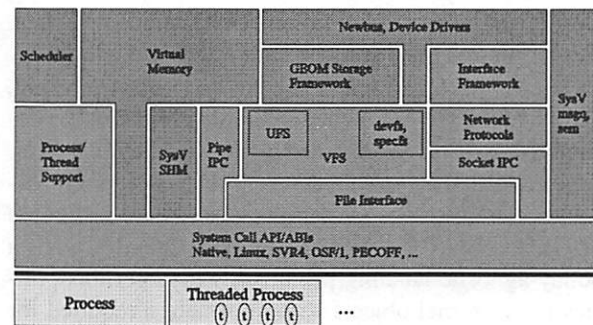


Figure 2: MAC Framework: Integration into Kernel Components

## 5.3 Entry Points

MAC Framework entry point invocations are conditionally compiled into kernel subsystems based on the configuration parameter `options MAC`. Several classes of entry points exist, including label management, event notification, decision functions, and access control checks. All entry points accept contextual information; typically this includes a subject process credential and a series of as objects, object label pointers, and call-specific arguments such as signal numbers or blocking disposition.

Some entry points, such as access control checks, return error values; other notification entry points are assumed always to succeed. Frequently, a set of related entry point invocations will be made around complex operations: for example, access control checks are required to create a new object in the file system namespace. Likewise, when label modifications occur, a two-phase commit is performed by the Framework to confirm that all policies will permit the relabel, and then to notify all policies to perform the actual operation.

Entry points are currently found in the cross-file system VFS code, device file system, mount/umount code, protocol-independent socket calls, pipe IPC code, BPF



packet sniffing code, IP fragment reassembly, IP socket send and receive code, network interface transmission and delivery, credential and process management code (including debugging, scheduling, signaling, and monitoring interfaces), kernel environmental variable management, kernel module management, per-architecture system calls, swap space management, and a variety of administrative interfaces such as time management, NFS service, `sysctl()`, and system accounting. These entry points permit policies to augment security decisions in a variety of forms.

## 5.4 Labels

While some system hardening models employ existing subject and object information (UNIX credential data, file permissions, ...) a number of important mandatory policies require additional subject and object labeling. For example, the MLS confidentiality policy makes decisions based on subject and object sensitivity labels: subjects are assigned clearances, and objects are assigned classifications. When policies require additional labels, the MAC Framework supports them through a policy-agnostic labeling primitive, which permits policies to tag kernel objects with information required for policy decision-making.

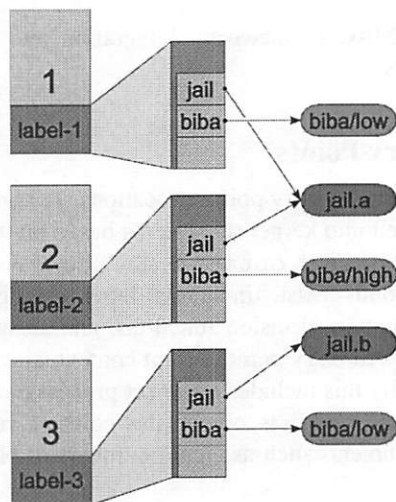


Figure 3: MAC Framework: Policy-Agnostic Label Storage

The label structure stored in kernel data structures is maintained by the MAC Framework: based on the life cycle of the data structure, the Framework provides per-object entry points for memory initialization, object allocation, and object destruction. The label structure consists an array of slots, each providing a union of a `void *` pointer and a `long`; slots are allocated to policies re-

quiring label storage for use in a policy-specific manner. Policy writers might choose to store an integer value, allocate per-label memory, or make use of referenced structures relying on the initialization and destruction calls to maintain reference counts. Initialization calls will often be used for memory allocation, and in some cases a blocking disposition will be passed as an argument to the call indicating whether blocking allocation is permitted; in these cases the initialization call is permitted to return a memory allocation failure, which will abort the allocation of the object.

Label storage is currently provided in the following kernel objects: BPF descriptors, process credentials, devfs directory entries, network interfaces, IP fragment reassembly queues (IPQ), sockets, pipes, mbufs, file system mount points, processes, and vnodes. A blocking disposition is provided for mbuf, socket, and IPQ initialization; if a failure occurs during label allocation, the mbuf and socket allocator code will return a memory exhaustion failure to the consumer. Unlike most other kernel objects, memory to hold the mbuf label is not stored within the mbuf structure itself: instead, it is stored in an `m_tag` hung off the mbuf header `m_tag` chain. Tags to hold MAC data will be allocated only when policies requiring MAC labels on mbufs are present in the system. This permits improved network performance of the MAC Framework in scenarios where flexible access control is required, but where mbuf labeling is not.

Additional support for persistent label storage is provided by any file system supporting extended attributes, including UFS1 and UFS2; while policies can determine whether and how the attributes are bound to policy-specific labels, the Framework constructs transactions to read, write, and cache vnode labels on supporting file systems.

This flexibility supports a wide variety of behaviors required for many interesting and useful access control policies.

## 5.5 Composition

Hardened or trusted systems are frequently shipped with a number of active (and hence composed) security policies. For example, many traditional "trusted" UNIX systems include the standard UNIX access control model, local discretionary extensions to that model (such as ACLs), the Multi-Level Security (MLS) confidentiality model protecting user data, and the Biba integrity model protecting the integrity of the Trusted Code Base (TCB) [3] [4]. Likewise, locally maintained security extensions are frequently deployed in combination with existing system security policies, forming cohesive (and ideally stronger) protection. As the MAC Framework is intended to assist vendors in combining security components to be deployed in a variety of environments, the



MAC Framework supports the simultaneous loading of several modules. While it may not be possible to coherently (or even safely) compose all access control policies, the MAC Framework provides a simple composition model that has proven useful in existing shipped systems: rights intersection. This composition largely maintains and assumes independence between the active policies, composing their behaviors only for two classes of operations:

- *Access control checks.* A precedence operator composes the results of an access control decision; the practical impact of this approach is that if any policy denies access to an object or operation, then the MAC Framework will return an access denial to the kernel service. However, the precedence operator also has the effect of sorting “Object not found” errors before “Access denied” errors, providing some useful precedence behavior when information flow policies are present.
- *User label requests.* Policies are permitted to deny access to relabel objects even if the label change request pertains only to label elements maintained by other policies. This has utility in a number of situations, including in the following example: the Biba integrity policy may forbid the changing of an MLS sensitivity label on a high integrity object by a low integrity subject, since the changing of a label might constitute an information flow operation from the perspective of the Biba policy.

The same composition model is employed to combine results from the native UNIX access control model and any models added using the MAC Framework. Currently, the task of determining whether two policies may be safely composed is left to the system designer or administrator, a reasonable requirement for many of the deployed environments of interest.

## 5.6 Policy Modules

Policies are typically encapsulated in a kernel module, although they may also be directly linked to the kernel. Policy modules consist of several elements (some optional):

- *Configuration.* Optional configuration parameters for the policy.
- *Policy logic.* Optional abstracted and centralized implementation of the policy’s access control logic.
- *Labeling.* Optional support for initializing, maintaining, and destroying labels on selected objects.
- *Label APIs.* Optional support for user process inspection and modification of labels on selected objects.
- *Access control.* Implementation of selected access control events that are of interest to the policy.

- *Policy events.* Optional implementation of policy initialization and destruction events.
- *Declaration.* Declaration of policy module identity, policy module properties, and registration of relevant policy operations.

## 5.7 Application Interfaces

The MAC Framework provides support for a number of classes of security-aware applications, including policy-agnostic or policy-aware labeling tools, and the login/user context management context routines. This is possible due to the policy-agnostic label management library and system calls, which allow applications to deal with MAC labels and elements in an abstract manner. The following functions are available to applications linked against the C library:

Retrieve the label of current or arbitrary process; set the current process label. `mac_get_pid()`, `mac_get_proc()`, `mac_set_proc()`.

Get and set file or pipe label by file descriptor. `mac_get_fd()`, `mac_set_fd()`.

Get and set file label by path; optionally follow symbolic links. `mac_get_file()`, `mac_set_file()`, `mac_get_link()`, `mac_set_link()`.

Execute a command and atomically modify the process label. `mac_execve()`.

Policy-specific system call multiplexor `mac_syscall()`.

Test for the presence of the MAC Framework or a specific policy `mac_is_present()`.

Convert labels to and from human-readable text `mac_from_text()`, `mac_to_text()`.

Allocate storage for a label appropriate to hold the specified label elements, or for a specific object based on system default label elements `mac_prepare()`, `mac_prepare_file_label()`, `mac_prepare_ifnet_label()`, `mac_prepare_process_label()`.

Release storage associated with a label `mac_free()`.

To support atomic change of label with execution events, `mac_execve()` provides an extension to the existing `execve()` system call accepting a requested target label. This is required to support the `execve_secure()` functionality used by the SEBSD port of FLASK/TE to FreeBSD from SELinux [11].

In addition, a general security policy entry point, `mac_security()` is provided so that policies may extend the set of system services without allocating new system call numbers.

## 6 Login Context Management

Many labeled access control policies assign user process labels on the basis of the identity of the user and

properties of the user account. Frequently this is performed in a role-based manner, where a set of available roles is assigned to a user, and then the user may select their active role from among the available roles. This assumes the ability to assign an initial label during the login process or when acting on behalf of the user, and then the ability to support constrained modification of the label based on the initial login configuration. The MAC Framework user process labeling APIs are sufficiently flexible to support this behavior.

For an initial pass at supporting automatic labeling at login, we extended the existing BSD login class database. The `master.passwd(5)` assigns one class to each user; a class may be shared by many users, and includes information such as the resource restrictions for the user, login and accounting properties, etc. We introduced two new fields:

Identifier	Description
label	The text form of the label to be assigned to user processes as part of the context management process.
ttylabel	The label to assign to the user's tty

The existing `setusercontext(3)` interface is extended to support a new flag `LOGIN_SETMAC`, indicating that the MAC label should be set as part of the login process. This flag is also implied by the `LOGIN_SETALL` flag used widely across programs setting user contexts. Process labeling tools, described in the next section, may then be used to update the process label subject to policy constraints. By instrumenting this one function and its relevant consumers, we were able to easily modify most key system daemons and applications to recognize the new process properties, including `sendmail(8)`, `cron(8)`, `login(1)`, `su(8)`, `ftpd(8)`, `inetd(8)` and others, making the changes relatively low impact. In the future, we may divorce the label selection database from the class database for the purpose of improved management, but this would not require changes to the user context API.

## 7 Application Integration

As most of the extensions policies of interest are mandatory policies, many applications that have specific adaptation to the system discretionary policy do not require changes for MAC. This occurs because objects created by processes will have labels automatically determined based on the process label or other process properties, rather than as application-provided arguments. The TrustedBSD MAC implementation ships with several tools to permit users to inspect and maintain labels on

objects, including:

Program	Description
<code>getpmac</code>	Inspect process MAC labels
<code>setpmac</code>	Set process MAC labels
<code>getfmac</code>	Inspect file MAC labels
<code>setfmac</code>	Set file MAC labels
<code>setfsmac</code>	Set file MAC labels based on a specification file

In addition, the following utilities were also modified to inspect and set MAC labels:

Program	Description
<code>ifconfig</code>	Inspect and set interface labels
<code>ps</code>	Inspect process labels
<code>ls</code>	Inspect file labels

Further extensions could easily be made to applications such as the KDE file system browser, Konqueror, to display and manage labels on file system objects.

## 8 Sample Policies

FreeBSD 5.0 ships with a number of sample policies—many appropriate for deployment in production systems. These demonstrate some of the scope of the capabilities of the MAC Framework, ranging from very simple un-labeled inter-process visibility protections to fully labeled policy environments such as Type Enforcement.

- `mac_biba`. Fixed-label hierarchal Biba integrity policy with compartments: assigns integrity labels to all system subjects and objects, then enforces an information flow policy based on limiting read-down and write-up operations.
- `mac_bsdextended`. File system firewall, maintains an access control rule list expressed in terms of UNIX credentials, file owners, and operation masks.
- `mac_ifoff`. Interface silencing policy, prohibiting unauthorized output on network interfaces—appropriate for use in environments where silent monitoring is required.
- `mac_lomac`. Floating label hierarchal Biba integrity policy based on the “Low watermark” scheme [7]: assigns integrity labels to all system subjects and objects, preventing write-up and forcing a subject downgrade on read-down.
- `mac_mls`. Fixed-label hierarchal Multi-Level Security confidentiality policy with compartments: assigns sensitivity labels to all system subjects and objects, then enforces an information flow policy based on limiting write-down and read-up operations.

- `mac_none`. Stub policy providing prototypes for all policy entry points—a starting point for new policies. Also useful for raw performance measurements without the cost of labeling and access control events.
- `mac_partition`. Simple labeled system partitioning policy, in which processes are assigned to system partitions and visibility of processes is limited based on the label of a process.
- `mac_portacl`. Add access control lists to control explicit IPv4 and IPv6 socket binding by protocol, port, and uid or gid.
- `mac_seeotheruids`. Simple system partitioning policy, in which process visibility is limited based on the UNIX credential of a process.
- `mac_test`. Policy to exercise the MAC Framework as well as test its invariants. Checks to make sure the MAC Framework is correctly managing the labels on objects, and instrumenting appropriate access control checks.
- `sebsd`. Port of the SELinux FLASK and TE implementations to FreeBSD, providing access to the FLASK security abstractions, Type Enforcement implementation, and adaptations of a mature system policy.

A broad scope of policies may be implemented using the MAC Framework; the Framework is structured so that policy authors may select what performance, security, and functionality trade-offs they wish to make in policy design, augmenting the system policy in ways that reflect local requirements. This flexibility makes the MAC Framework a useful tool in a broad variety of environments, reflecting the variety of deployment scenarios in which FreeBSD is used.

## 9 Performance Results

Three important performance goals were kept in mind during the design and implementation process for the TrustedBSD MAC Framework:

- Minimize performance impact of the MAC Framework on systems where it is disabled.
- Minimize the overhead of the MAC Framework on systems where it is enabled and possibly in use.
- Permit policy authors to make performance/security/complexity trade-offs local to their policy based on the requirements for the policy.

In this section, we explore some of the issues associated with performance measurement of the MAC Framework. The Framework is currently integrated into the FreeBSD 5.0-CURRENT development branch—as a result, current performance measurements are used to guide the development process and explore the even-

tual impact, rather than representing final performance results. Substantial effort has not yet been invested in fine-grained performance tuning, although initial measurements suggest performance well within the bounds of acceptability.

For each test, we consider several kernel configurations:

- **GENERIC**: Base-line kernel without MAC support.
- **MAC**: Kernel compiled with MAC support, but no active security policies.
- **MAC\_NONE**: One active “stub” policy, implementing all entry points but without additional locking or logic.
- **MAC\_BSDEXTENDED**: One active “file system firewall” policy, implementing file system access control entry points and making use of a locked policy.
- **MAC\_BIBA**: One active mandatory integrity policy, implementing comprehensive labeling and access control entry points for all system objects.

The GENERIC kernel permits us to explore baseline performance as a control for other configurations; MAC tests the overhead to simply include extensible security support in the system with no policies. The three sample policies allow us to consider the overhead of entering a policy module for each entry point (MAC\_NONE), the cost of unlabeled file system protections using a locked policy (MAC\_BSDEXTENDED), and the cost of a fully labeled system integrity policy touching most aspects of system operation (MAC\_BIBA).

These tests were run on a FreeBSD 5.0-CURRENT system from the `trustedbsd_mac` development branch from late March, 2003; tests were run on a single-processor 800MHz Intel PIII system with 128mb of memory and ATA 7200rpm 20gb hard disk. For file system related benchmarking, all writable file systems were recreated using the same geometry between tests since file system aging effects are not of interest for these tests; reboots occur between each test to flush storage-related caches and reset slab allocator and mbuf allocator state. All file systems use UFS2 for high performance meta-data storage.

### 9.1 Kernel Compile Throughput

In the `buildkernel` test, we perform a macro-benchmark focused on system throughput relying on effective CPU utilization, I/O performance, and file system meta-data performance. In this test, a FreeBSD kernel source tree is configured and built without modules (to reduce the I/O throughput dependency); time is measured in wall clock duration from start to finish. Lower execution times are preferred, indicating higher system throughput in completing the task.



The results of this test demonstrate a small but measurable performance change (0.1%) with MAC support. A slight relative increase in cost for the BSD/extended policy may be the result of acquiring a policy lock in order to process an access control decision; however, there is no statistically significant difference in performance between the various MAC policies and the base cost of the MAC Framework; UFS2 provides for high performance label access for the Biba policy.

## 9.2 Network Performance

The MAC Framework introduces security label structures into a variety of system data structures; of these, `struct mbuf` may be the most performance-sensitive. The `mbuf` structure provides for optimized network management, and has been the subject of substantial prior performance work, providing optimized packet construction and parsing, copy-on-write semantics, zero-copy semantics, and fragmentation management. From the perspective of MAC policy modules, only header `mbufs` are of interest, as they represent the header for a network packet or datagram. In our first pass implementation, we inserted a `struct label` directly into the `m_pkthdr` data structure; as the implementation evolved, the `m_tag` meta-data service became available on FreeBSD; this service permits chaining of arbitrary meta-data onto `mbuf` headers without modification of the base structure.

In the `m_pkthdr` approach, all kernels pay a memory overhead for labeling support, although kernels without `options MAC` do not pay the label life cycle costs. With the `m_tag` approach, only kernels with `options MAC` pay the memory overhead, although we presupposed that there would be a higher cost for using tags for label storage due to greater administrative overhead in maintaining lists and allocating storage. To optimize the `m_tag` approach, we implemented lazy tag allocation: tags are only allocated to hold label data when a policy expresses interest in labeling `mbuf` headers.

We consider two tests from the `netperf` suite: `UDP_RR` and `UDP_STREAM`, which respectively test the per-transaction cost of a Request/Receive RPC, and raw network throughput. The request/response test measures the throughput of the system relative to synchronous one-byte packets between a client and a server, and is intended to measure the performance impact of a change in terms of number of packets transferred. The stream test uses a larger packet size and does not synchronously wait for a response before continuing, generally measuring the performance impact of a change in terms of data transferred.

In Figure 5, the performance cost per-packet is illustrated: the introduction of MAC support produces a measurable change; depending on the strategy for labeling

`mbufs`, that change varies substantially. With inclusion of the label directly in the `mbuf` header, an 11.5% performance overhead is accepted for enabling MAC support. Adding the stub policy increases that cost to 12.2%; adding a complex labeled policy performing per-label memory allocation, such as Biba, increases that drop to 14.9% of the GENERIC packet throughput.

With lazy `m_tag` labeling, the performance trade-off is changed: the cost of introducing MAC is 4.8% (substantially less than using `mbuf` headers); with a stub policy implementing `mbuf` label entry points but not allocating labels, that cost increases to 8.5% (also less than `mbuf` headers). However, performance with a Biba performance is reduced by 17.1%, showing an increased cost for heavily labeled policies such as Biba.

The second set of trade-offs best fits the needs of the TrustedBSD Project: minimize overhead for MAC-disabled systems, and permit a performance/complexity cost decision by policy authors.

In Figure 6, a similar pattern emerges, where-in the introduction of MAC support results in a 6.1% throughput penalty. Use of a stub policy increases that cost to 7.7%; Biba labeling increases the cost to 10.3%. However, with lazy `m_tag` labeling, the base cost of MAC support is reduced to 3.7%; the stub policy increases this cost to 4.4%, and with Biba to 9.7%. The proportionally lower performance cost with this test derives from the reduced relative overhead resulting from reduced packet counts relative to data transferred.

Again, lazy `m_tag` allocation better meets our requirements by permitting better non-MAC performance with a more clear performance trade-off for complexity. Unlike the packet count testing, performance for the Biba policy actually increases with lazy `m_tags` relative to `mbuf` headers, a result that we attribute to differences in the caching and allocation policies for the UMA Slab Allocator versus the `mbuf` allocator in handling memory clearing for new allocations.



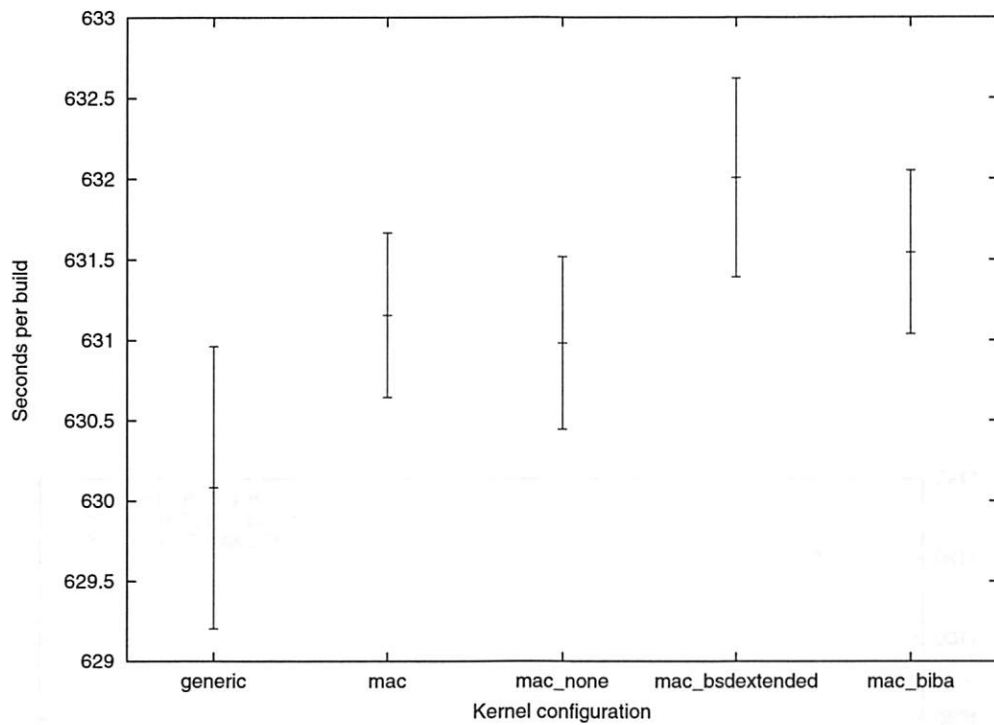


Figure 4: Time to make `buildkernel` with various kernel configurations (lower is better).

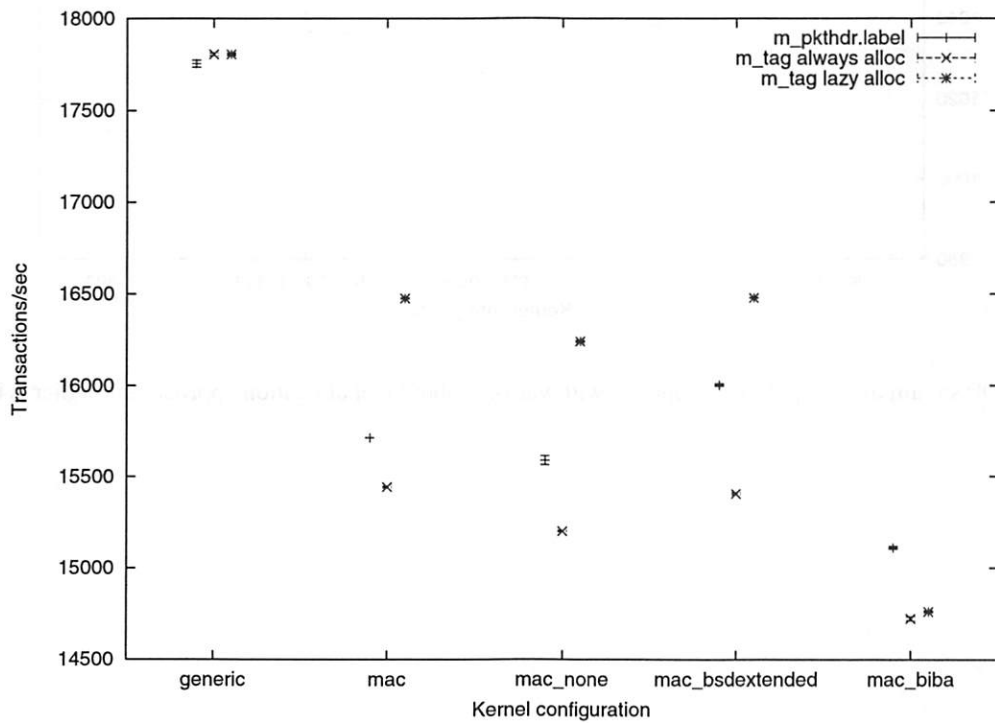


Figure 5: UDP request/response throughput over loopback with various mbuf label allocation approaches (higher is better).

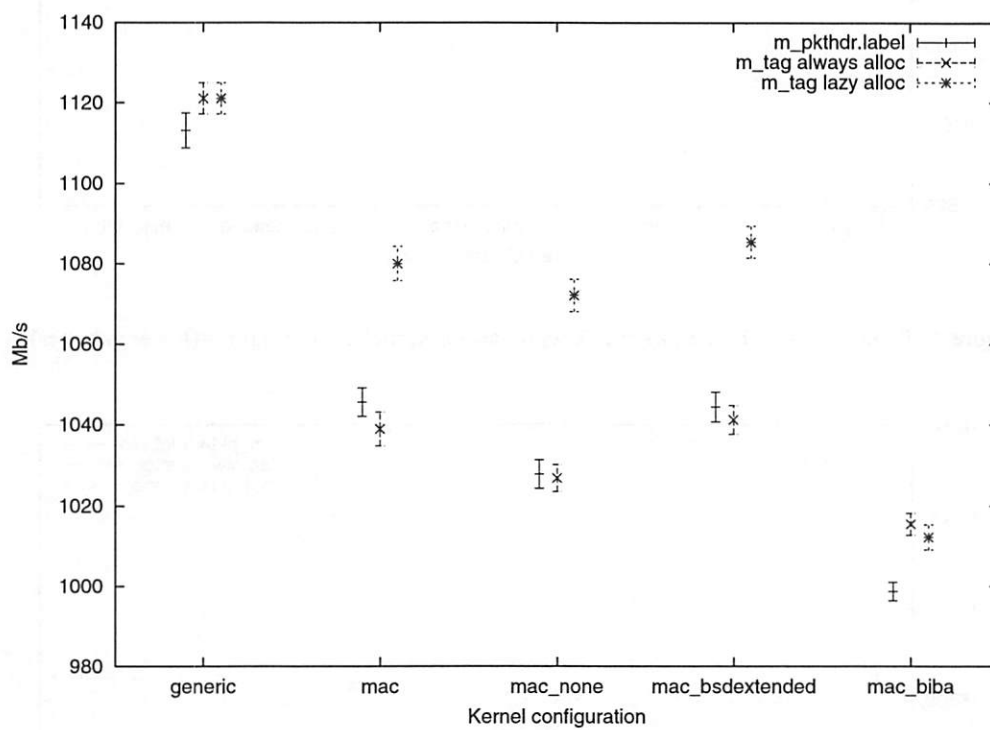


Figure 6: UDP stream throughput over loopback with various mbuf label allocation approaches (higher is better).

## 10 Related Work

Substantial prior and current work exists relating to kernel access kernel and extensibility research.

In the area of prior deployed systems, “Trusted” variants of most commercial UNIX platforms exist, including Trusted Solaris, and Trusted IRIX [14]. In addition, there are a number of third party security extension products that exist for these systems, including Argus’s Pit-Bull product, which provides a product alternative with many of the same features [2]. These products largely rely on Multi-Level Security (MLS) [3] and Biba integrity [4] to provide mandatory data confidentiality and TCB integrity; earlier TrustedBSD work has focused on implementing support in FreeBSD for these models using similar integration approaches [17][18][19]. TrustedBSD MAC policy modules exist expressing both MLS and Biba in functionally similar forms.

In the area of access control extensibility, early work included the Generalized Framework for Access Control (GFAC), which proposes a separation of policy and enforcement [1]; this model is implemented in Linux in RSBAC [13].

The FLASK framework provides for similar types of separation of policy and enforcement, although with higher level labeling abstraction in the form of a security ID (SID) and a focus on Linux Security Modules (LSM) provides a set of kernel extension hooks to facilitate integration of systems such as SELinux without committing the Linux operating system to a particular model [16]. LSM provides a void pointer for label storage in each supported kernel object, and has access control notions similar to the TrustedBSD MAC Framework. However, the semantics of the hooks are weaker, and the LSM framework does not provide for policy composition and persistent labeling, relying on policy modules to implement these services. Type Enforcement for policy representation [15][11]. A prototype port of the SELinux FLASK and TE implementations has been made to layer on top of the TrustedBSD MAC Framework via the SEBSD policy module.

## 11 Future Work

Future work on the MAC Framework will likely fall into a number of areas:

- Improve the completeness and expressiveness of the MAC Framework; increase the number of kernel objects and methods that are protected by the Framework to permit broader protections.
- Mature the experimental policy modules.
- Continue to adapt and merge the SEBSD policy module to run properly with FreeBSD: in particular, determine how best to satisfy the differing requirements of SEBSD and most other policies re-

garding process label transitions.

- Continue porting the MAC Framework and its policies to Darwin and Mac OS X.

## 12 Conclusion

The TrustedBSD MAC Framework provides a generalized mechanism by which the FreeBSD kernel security model can be augmented at run-time. Along with the framework, we have also implemented a number of security extension modules that rely solely on the framework to interface with existing kernel abstractions. This separation of security extensions from the actual kernel implementation of services improves the capacity for third party providers to develop and ship system security extensions by lowering the cost to develop and maintain the extensions. Through a simple composition model, it is possible to perform a limited set of “useful” compositions of security extensions. Preliminary performance measurement illustrates a measurable but small performance cost for the framework and many policy modules. the Framework permits policy authors to select complexity and performance trade-offs based on local requirements, supporting both simple hardening policies and complex information flow policies.

## 13 Acknowledgments

This research was supported under DARPA/SPAWAR contract N66001-01-C-8035 (“CBOSS”).

The authors would like also to thank the anonymous reviewers of this paper, as well as other contributors to the TrustedBSD Project, including Chris Faulhaber, Ilmar Habibulin, Adam Migus, and Thomas Moestl. The authors would also like to thank Angelos Keromytis and Erez Zadok for their shepherding of this paper.

## 14 Availability

The TrustedBSD MAC Framework is available under a two-clause BSD license, making it appropriate for open and closed-source, research, educational or commercial use without restriction. It is included in FreeBSD 5.0, as an experimental feature, and will mature over the FreeBSD 5.x life time. More information may be found at:

<http://www.FreeBSD.org/>

<http://www.TrustedBSD.org/>

## 15 Bibliography

### References

- [1] M. D. Abrams, K. W. Eggers, L. J. L. Padula, and I. M. Olson. A generalized framework for access control: An informal description. In *Proc. 13th NIST-NCSC National Computer Security Conference*, pages 135–143, 1990.

- [2] Argus products overview: Pitbull. <http://www.argussystems.com/product/overview/pitbull/>.
- [3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corp., Bedford MA, May 1973.
- [4] K. Biba. Integrity constraints for secure computer systems, 1977.
- [5] N. C. C. I. Board. Common criteria version 2.1 (ISO IS 15408), 2000.
- [6] Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *RSP: 21th IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.
- [7] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.
- [8] FreeBSD Project. FreeBSD home page. <http://www.FreeBSD.org/>.
- [9] N. S. A. Information Systems Security Organization. Controlled access protection profile version 1.d, October 1999.
- [10] N. S. A. Information Systems Security Organization. Labeled security protection profile version 1.b, October 1999.
- [11] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. Technical report, U.S. National Security Agency (NSA), Oct. 2000.
- [12] U. S. D. of Defense. *Trusted Computer System Evaluation Criteria*. Department of Defense, December 1985.
- [13] Rule set based access control (RSBAC) for linux. <http://www.rsbac.org/>.
- [14] SGI. B1 sample source code. <http://oss.sgi.com/projects/ob1/>.
- [15] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *8th USENIX Security Symposium*, pages 123–139, Washington, D.C., USA, Aug. 1999. USENIX.
- [16] G. S. Support. Linux security modules:.
- [17] TrustedBSD Project. TrustedBSD home page. <http://www.TrustedBSD.org/>.
- [18] R. Watson. Introducing supporting infrastructure for trusted operating system support in FreeBSD. In *BSD Conference*, Monterey, CA, USA, October 2000.
- [19] R. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the USENIX Annual Technical Conference*, June 2001.



# Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel

Andrea Arcangeli

SuSE Labs

Mingming Cao, Paul E. McKenney, and Dipankar Sarma

IBM Corporation

## Abstract

Read-copy update (RCU) allows lock-free read-only access to data structures that are concurrently modified on SMP systems. Despite the concurrent modifications, read-only access requires neither locks nor atomic instructions, and can often be written as if the data were unchanging, in a “CS 101” style. RCU is typically applied to read-mostly linked structures that the read-side code traverses unidirectionally.

Previous work has shown no clear best RCU implementation for all measures of performance. This paper combines ideas from several RCU implementations in an attempt to create an overall best algorithm, and presents a RCU-based implementation of the System V IPC primitives, improving performance by more than an order of magnitude, while increasing code size by less than 5% (151 lines). This implementation has been accepted into the Linux 2.5 kernel.

## 1 Introduction

The past two years have seen much discussion of RCU, along with the design and coding of a number of implementations and uses of RCU, one of which is now part of the Linux 2.5 kernel [Sarma02].

Comparisons with other concurrent update mechanisms [McK01b, Linder02a] have shown that RCU can greatly simplify and improve performance of code accessing read-mostly linked-list data structures. This paper adds a performance evaluation of RCU applied to the Linux System-V IPC primitives. RCU can also improve performance of code modifying linked-list structures when there is a high system-wide aggregate update rate across all such structures [McK98a].

Comparison of multiple RCU implementations [McK02a] showed, as noted in the abstract, that there is no overall best algorithm. The *rcu-poll* algorithm had the shortest latency, while the *rcu-timer* algorithm had the lowest overhead. This paper presents a parallelized

variant of *rcu-poll* in an attempt to gain the best of both worlds.

Section 2 provides background on RCU, Section 3 reviews attempts to produce a single best RCU implementation, Section 4 describes an RCU-based implementation Linux’s System V IPC primitives, and Section 5 describes future plans.

## 2 Background

This section gives a brief overview of RCU; more details are available elsewhere [McK98a, McK01b, McK02a]. Section 2.1 presents a visual example of RCU-based list manipulation, Section 2.2 contains a glossary of RCU-related terms, Section 2.3 presents concepts, Section 2.4 presents the RCU API, and Section 2.5 illustrates an analogy between RCU and reader-writer locking.

### 2.1 Example

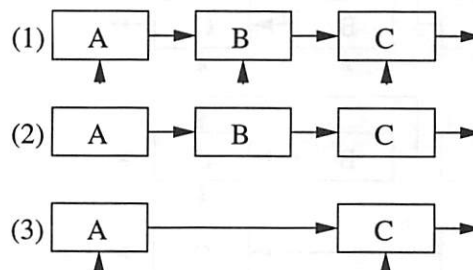


Figure 1: Lock Protecting Deletion and Search

On SMP systems, any searching of or deletion from a linked list must be protected by a lock. When element B is deleted from the list shown in Figure 1, searching code is guaranteed to see this list in either the initial state (1) or the final state (3). In state (2), when element B is being deleted, the reader-writer lock guarantees that no readers (indicated by the triangles) will be accessing the list.

However, many lists are searched much more often than they are modified. For example, an IP routing table would normally change at most once per few minutes,

---

The views expressed in this paper are the authors' only, and should not be attributed to SuSE or IBM.

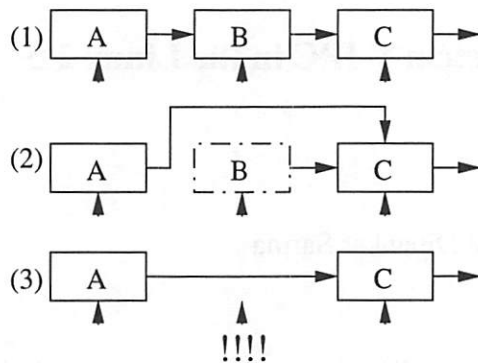


Figure 2: Race Between Deletion and Search

but might be searched many thousands of times per second. This could result in well over a million accesses per update, making lock-acquisition overhead burdensome to searches.

Unfortunately, omitting locking when searching means that the update no longer appears to be atomic. Instead, the update takes the multiple steps shown in Figure 2. A search might be referencing element B just as it was freed up, resulting in crashes, or worse, as indicated by the reader referencing nothingness in step (3).

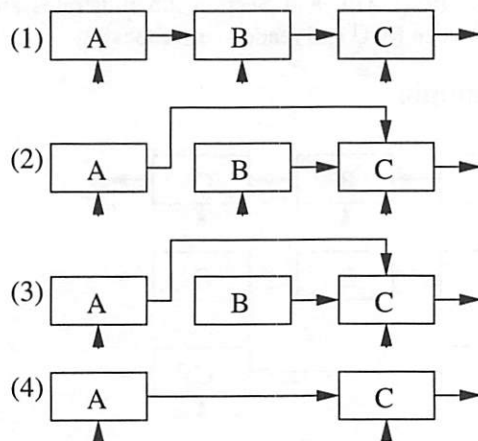


Figure 3: RCU Protecting Deletion and Search

One solution to this problem is to delay freeing up element B until all searches have given up their references to it, as shown in Figure 3. RCU indirectly determines when all references have been given up. To see how this works, recall that there are normally restrictions on what operations may be performed while holding a lock. For example, in the Linux kernel, it is forbidden to do a context switch while holding any spinlock. RCU mandates these same restrictions: even though the RCU-protected search need not acquire any locks, it is forbidden from performing any operation that would be forbidden if it

were in fact holding a lock.

Therefore, any CPU that is seen performing a context switch after the linked-list deletion shown in step (2) of Figure 3 cannot possibly hold a reference to element B. As soon as all CPUs have performed a context switch, there can no longer be any readers, as shown in step (3). Element B may then be safely freed, as shown in step (4).

A simple, though inefficient, RCU-based deletion algorithm could perform the following steps in a non-preemptive Linux kernel (preemptive kernels can be handled as well [McK02a]):

1. Unlink element B from the list, but do *not* free it. The state of the list will be that shown in step (2) of Figure 3.
2. Run on each CPU in turn. At this point, each CPU has performed one context switch after element B has been unlinked. Thus, there cannot be any more references to element B.
3. Free up element B.

Much more efficient implementations have been described elsewhere [McK98a, McK02a]. The following sections present the concepts underlying RCU.

## 2.2 Glossary

The following definitions help illuminate the concepts underlying RCU.

**Live Variable:** A variable that might be accessed before it is next modified, so that its current value has some possibility of influencing future execution state.

**Dead Variable:** A variable that will be modified before it is next accessed, so that its current value cannot possibly have any influence over future execution state.

**Critical Section:** A region of code that is protected from outside interference by some locking mechanism, such as spinlocks or RCU.

**Read-Side Critical Section:** A region of code that is protected from other CPUs' modifications, but which allows multiple CPUs to read simultaneously.

**Temporary Variable:** A variable that is only live inside a critical section. One example is an auto variable used as a pointer while traversing a linked list.

**Permanent Variable:** A variable that is live outside of critical sections. One example would be the header for a linked list. Although it is possible for the same variable to be temporary sometimes and permanent at other times, this practice can lead to confusion, so is not generally recommended. Relying on the register-allocation capabilities of modern optimizing compilers is usually a far better strategy.

**Quiescent State:** A point in the code where all of the current CPU's temporary variables that were previously in use in a critical section are dead. In a non-preemptive Linux kernel, a context switch is a quiescent state for CPUs. In a preemptive Linux kernel, `rcu_read_lock()` and `rcu_read_unlock()` suppress preemption for short read-side critical sections, so that context switch is still a quiescent state with respect to these read-side critical sections. Although there are implementations of RCU that do not require preemption to be suppressed [Gamsa99, McK02a], they can be prone to excessively long grace periods.

**Grace Period:** Time interval during which all CPUs pass through at least one quiescent state. The key property of a grace period is that the values that were contained in any temporary variable in use in a critical section at the beginning of the grace period cannot possibly have any direct effect after the end of the grace period. This property will be examined more closely in the next section. Note that any time interval containing a grace period is itself a grace period.

## 2.3 Concepts

Without special action in the deletion code, the search code would be prone to races with those deletions, described in Section 2.1. To handle such race conditions, the update side performs the update in two phases as follows: (1) remove permanent-variable pointers to the item being deleted, and (2) after a grace period has elapsed, free up the item's memory.

The grace period is not a fixed time duration, but is instead inferred by checking for per-CPU quiescent states, such as context switches in non-preemptive environments. Since kernel threads are prohibited from holding locks across a context switch, they also prohibited from holding pointers to data structures protected by those locks across context switches—after all, the entire data structure could well be deleted by some other CPU at any time this CPU does not hold the lock.

A trivial implementation of RCU in a non-preemptive kernel could simply declare the grace period over once it observed each CPU undergoing a context switch. Once the grace period completes, no CPU references the deleted item, and no CPU can gain such a reference. It is therefore safe to free up the deleted item.

With this approach, searches already in progress when the first phase executes might (or might not) see the item being deleted. However, searches that start after the first phase completes are guaranteed to never reference this item. Efficient mechanisms for determining the duration of the grace period are key to RCU, and are described fully elsewhere [McK02a]. These mechanisms

```
void synchronize_kernel(void);
void call_rcu(struct rcu_head *head,
              void (*func)(void *arg),
              void *arg);
struct rcu_head {
    struct list_head list;
    void (*func)(void *obj);
    void *arg;
};
void rcu_read_lock(void);
void rcu_read_unlock(void);
```

Figure 4: RCU API

track “naturally occurring” quiescent states, which removes the need for adding expensive context switches. In addition, these mechanisms take advantage of the fact that a single grace period can satisfy multiple concurrent updates, amortizing the cost of detecting a grace period over the updates.

## 2.4 RCU API

Figure 4 shows the external API for RCU. The `synchronize_kernel()` function blocks for a full grace period. This is a simple, easy-to-use function, but imposes expensive context-switch overhead on its caller. It also cannot be called with locks held or from softirq and irq (interrupt) contexts.

Another approach, taken by `call_rcu()`, is to schedule a callback function `func` to be called with argument `arg` after the end of a full grace period. Since `call_rcu()` never sleeps, it may be called with locks held. It may also be called from the softirq and irq contexts. The `call_rcu()` function uses its `struct rcu_head` argument to remember the specified callback function (in the `func` field) and argument (in the `arg` field) for the duration of the grace period. At the end of the grace period, the RCU subsystem invokes the function pointed to by the `func` field, passing it the contents of the `arg` field. An `rcu_head` is often placed within a structure being protected by RCU, eliminating the need to separately allocate it.

The primitives `rcu_read_lock()` and `rcu_read_unlock()` demark a read-side RCU critical section, but generate no code in non-preemptive kernels. In preemptive kernels, they disable preemption within the critical section, which is required because both `synchronize_kernel()` and `call_rcu()` declare the grace period to be over once each CPU has completed a context switch. Suppressing preemption during the read-side RCU critical section prevents the crashes that could result from such prematurely ending grace periods. There is a patch that implements a proposed `call_rcu_preempt()` [McK02a] that tolerates preemption in read-side critical sections (based on the K42 implementation [Gamsa99]), but at this writing, there are no RCU uses envisioned in Linux

```
list_add_rcu(struct list_head *new,
            struct list_head *head);
list_add_rcu_tail(struct list_head *new,
                  struct list_head *head);
list_del_rcu(struct list_head *entry);
list_for_each_rcu(struct list_head *pos,
                  struct list_head *head);
list_for_each_safe_rcu(struct list_head *pos,
                       struct list_head *n,
                       struct list_head *head);
```

Figure 5: RCU List API

```
1 struct el {
2     struct list_head list;
3     long key;
4     long data;
5     struct rcu_head my_rcu_head;
6 };
```

Figure 6: List Element Data Structure

that could tolerate the extremely long grace periods that might result from preemption on a busy system.

Modern microprocessors, particularly the DEC/Compaq/HP Alpha, feature very weak memory consistency models. These models require use of special “memory barrier” instructions. However, since proper use of these instructions is often difficult to understand and even more difficult to build good test suites for, there is an extension to the Linux list-manipulation API for use with RCU, as shown in Figure 5. Each primitive in this extension is equivalent to its non-RCU counterpart, but with the addition of whatever memory-barrier instructions are required on the machine in question [Spraul01].

RCU may be applied to data structures other than lists, but in such cases, memory-barrier instructions must be used explicitly. An example of such a situation is shown in Section 4.5.

## 2.5 Reader-Writer-Locking/RCU Analogy

Although RCU has been used in a great many interesting and surprising ways, one of the most straightforward is as a replacement for reader-writer locking. In this section, we present this analogy, protecting the simple doubly-linked-list data structure shown in Figure 6 with reader-writer locks and then with RCU, comparing the results. This structure has a `struct list_head` that is manipulated by the standard Linux list-manipulation primitives, a search key, and a single integer for data. The RCU primitive `call_rcu()` requires some space in the data element, which is supplied by the `my_rcu_head` field.

The reader-writer-lock/RCU analogy substitutes primitives as shown in Table 1. The asterisked primitives are no-ops in non-preemptible kernels; in preemptible kernels, they suppress preemption, which

is an extremely cheap operation on the local task structure. Note that since neither `rcu_read_lock()` nor `rcu_read_unlock()` block irq or softirq contexts, it is necessary to add primitives for this purpose where needed. For example, `read_lock_irqsave` must become `rcu_read_lock()` followed by `local_irq_save()`.

Reader-Writer Lock	Read-Copy Update
<code>rwlock_t</code>	<code>spinlock_t</code>
<code>read_lock()</code>	<code>rcu_read_lock() *</code>
<code>read_unlock()</code>	<code>rcu_read_unlock() *</code>
<code>write_lock()</code>	<code>spin_lock()</code>
<code>write_unlock()</code>	<code>spin_unlock()</code>
<code>list_add()</code>	<code>list_add_rcu()</code>
<code>list_addtail()</code>	<code>list_add_tail_rcu()</code>
<code>list_del()</code>	<code>list_del_rcu()</code>
<code>list_for_each()</code>	<code>list_for_each_rcu()</code>

\* no-op unless CONFIG\_PREEMPT, in which case preemption is suppressed

Table 1: Reader-Writer-Lock/RCU Substitutions

Deletion from the list is illustrated by the upper section of Table 2. The `write_lock()` and `write_unlock()` are replaced by `spin_lock()` and `spin_unlock()`, respectively, the `list_del()` is replaced by `list_del_rcu()`, and `my_free()` is replaced by `call_rcu()`. The `call_rcu()` will, after a grace period elapses, pass `p` to function `my_free()`, using the `struct rcu_head` in the `my_rcu_head` field to keep track of the deferred function call and argument.

Insertion into the list is illustrated by the second section of Table 2. Again, the `write_lock()` and `write_unlock()` are replaced by the simple `spin_lock()` and `spin_unlock()`, respectively. The `list_addtail()` is replaced by `list_add_tail_rcu()`. The rest of the code remains the same.

Searching the list is illustrated by the third section of Table 2. Locking is handled by the caller, so the two variants differ only in that the `list_for_each()` is replaced by `list_for_each_rcu()`.

Searching the list for read-only access is illustrated by the last section of Table 2. The only difference is that the `read_lock()` and `read_unlock()` primitives are replaced by `rcu_read_lock()` and `rcu_read_unlock()`, respectively. Again, the bulk of the code remains the same for both cases.

Although this analogy can be quite compelling and useful, there are some caveats:

1. Read-side critical sections may see “stale data,” that has been removed from the list but not yet



Reader-Writer Lock	Read-Copy Update
<pre> 1 void delete(long mykey) 2 { 3     struct el *p; 4     write_lock(&amp;list_lock); 5     p = search(mykey); 6     if (p != NULL) { 7         list_del(p); 8     } 9     write_unlock(&amp;list_lock); 10    my_free(p); 11 } </pre>	<pre> 1 void delete(long mykey) 2 { 3     struct el *p; 4     spin_lock(&amp;list_lock); 5     p = search(mykey); 6     if (p != NULL) { 7         list_del_rcu(p); 8     } 9     spin_unlock(&amp;list_lock); 10    call_rcu(&amp;p-&gt;rcuhead, 11            (void (*)(void *))my_free, p); 12 } </pre>
<pre> 1 void insert(long key, long data) 2 { 3     struct el *p; 4     p = kmalloc(sizeof(*p), GPF_ATOMIC); 5     p-&gt;key = key; 6     p-&gt;data = data; 7     write_lock(&amp;list_lock); 8     list_add_tail(&amp;(p-&gt;list), &amp;head); 9     write_unlock(&amp;list_lock); 10 } </pre>	<pre> 1 void insert(long key, long data) 2 { 3     struct el *p; 4     p = kmalloc(sizeof(*p), GPF_ATOMIC); 5     p-&gt;key = key; 6     p-&gt;data = data; 7     spin_lock(&amp;list_lock); 8     list_add_tail_rcu(&amp;(p-&gt;list), &amp;head); 9     spin_unlock(&amp;list_lock); 10 } </pre>
<pre> 1 struct el *search(long mykey) 2 { 3     struct el *p; 4     list_for_each(p, &amp;head) { 5         if (p-&gt;key == mykey) { 6             return (p); 7         } 8     } 9     return (NULL); 10 } </pre>	<pre> 1 struct el *search(long mykey) 2 { 3     struct el *p; 4     list_for_each_rcu(p, &amp;head) { 5         if (p-&gt;key == mykey) { 6             return (p); 7         } 8     } 9     return (NULL); 10 } </pre>
<pre> 1 /* Read-only search */ 2 struct el *p; 3 read_lock(&amp;list_lock); 4 p = search(mykey); 5 if (p == NULL) { 6     /* handle error condition */ 7 } else { 8     /* access *p w/out modifying */ 9 } 10 read_unlock(&amp;list_lock); </pre>	<pre> 1 /* Read-only search */ 2 struct el *p; 3 rcu_read_lock(); /* nop unless CONFIG_PREEMPT */ 4 p = search(mykey); 5 if (p == NULL) { 6     /* handle error condition */ 7 } else { 8     /* access *p w/out modifying */ 9 } 10 rcu_read_unlock(); /* nop unless CONFIG_PREEMPT */ </pre>

Table 2: Reader-Writer-Lock and Read-Copy-Update Analogy

freed. There are some situations (e.g., routing tables for best-effort protocols) where this is not a problem. In other situations, such stale data may be detected and rejected [Pugh90], as illustrated in Section 4.

2. Read-side critical sections may run concurrently with write-side critical sections.
3. The grace period will delay freeing of memory, which means that both the memory and the cache footprint of the code will be somewhat larger when using RCU than when using reader-writer locking.
4. When changing to RCU, write-side reader-writer locking code that modifies list elements in place must often be restructured to prevent read-side RCU code from seeing the data in an inconsistent state. In many cases, this restructuring will be quite straightforward, for example, creating a new list element with the desired state, then replacing the old element with the new.

Where it applies, this analogy can deliver full parallelism with almost no increase in complexity. For example, Section 4 shows how applying this analogy to System V IPC yields order-of-magnitude speedups with a very small increase in code size and complexity.

RCU has also been used as a lazy barrier synchronization mechanism, as a mode-change control mechanism, as well as for more sophisticated list maintenance. Retrofitting existing code with RCU as shown above can produce significant performance gains, but of course the best results are obtained by designing RCU into the algorithms and code from the start.

Other methods have been proposed for eliminating locking [Herlihy93], and, when combined with more recent refinements [Michael02a, Michael02b], these methods are practical in some circumstances. However, they still require expensive atomic operations on shared storage, resulting in pipeline stalls, cache thrashing, and memory contention, even for read-only accesses.

### 3 RCU Implementation

The performance of RCU is critically dependent on an efficient implementation of the `call_rcu()` primitive. The more efficient the implementation, the greater the number of situations that RCU may profitably be applied to.

However, grace-period latency is also an important measure of performance – while CPU overhead negates the performance benefits of RCU, excessively long grace-period latencies can result in all available memory queued up waiting for a grace period to end. However, the conditions that cause excessively long grace-period latencies have other bad effects, even in absence of RCU, such as grossly degraded response times.

A detailed description of these algorithms has been presented elsewhere [McK02a], but a brief description of each follows. The *rcu\_poll* algorithm uses IPIs to force each CPU to quickly enter a quiescent state, which results in grace periods of much less than a millisecond on idle systems. However, the resulting increased scheduler overhead can outweigh the performance benefits of RCU. The *rcu\_ltimer* algorithm instruments the `scheduler_tick()` function that is called with HZ frequency on each CPU, resulting in extremely low overhead, but with grace periods in excess of 100 milliseconds. The *rcu\_sched* algorithm uses a token-passing scheme that holds the promise of extremely low overheads (which are currently masked by the cache thrashing of a global counter), but can have grace periods of almost one minute in duration. It is likely that the *rcu\_sched()* algorithm can be reworked to eliminate these disadvantages, which may result in it being the best overall algorithm. However, the *rcu\_ltimer* is the best match for current RCU uses in the Linux kernel, so this algorithm is implemented in the 2.5 Linux kernel.

The best grace-period latencies were obtained using *rcu\_poll*, which invokes `resched_task()` on each CPU to force context switches, resulting in latencies more than an order of magnitude shorter than those of *rcu\_ltimer* and *rcu\_sched*, as shown in Figure 7. This benchmark was run on an 8-CPU 700MHz Intel Xeon system with 1MB L2 caches and 6GB of memory using the *dcache-rcu* patch [LSE].

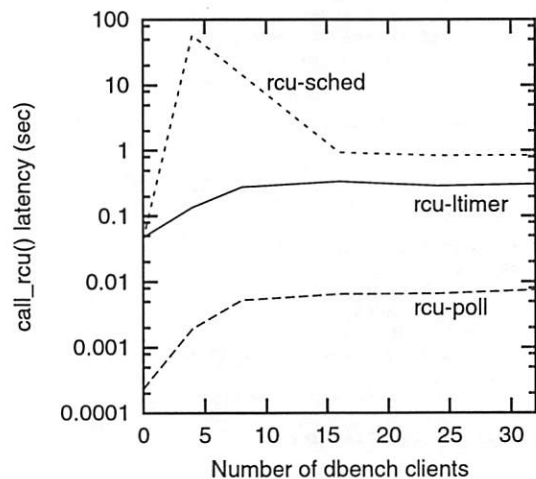


Figure 7: `call_rcu()` Latency Under dbench Load (logscale)

However, *rcu\_poll*'s single global callback list results in cache thrashing and high overhead, and it obtains the short grace-period latencies by frequently invoking

the scheduler, the combination of which at times overwhelms the performance benefits of RCU. The cache thrashing is caused by: (1) enqueueing onto this single list from multiple CPUs and (2) callbacks running on a CPU other than the one that registered them. This latter effect causes data structures processed by the callback to be pulled from the registering CPU to the CPU running the callback.

A modified *rcu-poll* algorithm was constructed having per-CPU lists of callbacks. This eliminates both sources of cache thrashing: each CPU manipulates only its own list and callbacks always run on the CPU that registered them. The frequent scheduler invocation remains, as this is required to obtain the excellent `call_rcu()` latencies.

Table 3 compares the performance of *rcu-ltimer* (in Linux 2.5 kernel), *rcu-sched*, and the parallelized version of *rcu-poll*. These results show that *rcu-ltimer* completes each iteration slightly (but statistically significantly) more quickly than does *rcu-poll*, and with 8.6% less CPU utilization. They also show *rcu-sched* completes each iteration as fast as does *rcu-ltimer*, but with 5.7% more CPU utilization. This benchmark was run on a 4-CPU PIII Intel Xeon with 1MB L2 cache and 1GB of memory. Profile results show that *rcu-poll* is incurring significant overhead in the scheduler and in its `force_cpu_reschedule()` function, indicating that although its cache-thrashing behavior has been addressed, *rcu-poll* is buying its excellent grace-period latency with significantly increased overhead. The *rcu-sched* implementation is unchanged; future work includes optimizing it to eliminate cache thrashing and atomic instructions in order to reduce its overhead.

	CPU Utilization		ms/Iteration	
	Avg	Std	Avg	Std
<i>rcu-ltimer</i> (2.5)	77.52%	0.05%	22.47	0.01
<i>rcu-poll</i>	84.20%	0.13%	22.95	0.03
<i>rcu-sched</i>	81.95%	0.20%	22.46	0.02

Table 3: dcachebench Comparison

Therefore, unless grace-period latency is of paramount concern, the *rcu-ltimer* implementation of RCU should be used. Should latency become a critical issue in the future, we will investigate modifications to improve the latency of *rcu-ltimer*.

An optimized *rcu-sched* might beat *rcu-ltimer*'s overhead. If this is the case, reduction of grace-period latency would become a considerably more urgent matter.

## 4 RCU Implementation of System V IPC

This section describes how the reader-writer-lock/RCU analogy described in Section 2.5 was used to break up

the global locks used by Linux's System V IPC primitives. These locks guard the following: (1) mapping from IPC identifiers to corresponding `kern_ipc_perm` structures, (2) expanding the mapping arrays, and (3) individual IPC operations. A straightforward modification would replace these global locks with reader-writer locks, allowing mapping operations to be performed in parallel.

However, we took the additional step of following the analogy, replacing the global locks with RCU [Cao02] to guard the mapping arrays and per-`kern_ipc_perm` locks to guard the IPC operations, which resulted in significant system-level speedups on database benchmarks. This modification also serves to illustrate use of explicit memory barriers and use of a deleted flag to prevent access to stale data.

The remainder of this section focuses on the changes to the System V semaphores; analogous changes were made to message queues and shared memory.

### 4.1 Semaphore Data Structures

The semaphore data structures are shown in Figure 8. The global `ipc_ids` structure tracks the state of all semaphores currently in use. Among other things, it contains a global lock `ary` and a pointer `entries` that points to an array of pointers of `ipc_id` structures. Each such entry is either NULL or points to a `sem_array` structure, which represents a set of semaphores that has been created by a single `semget()` system call. The array of `ipc_id` structures is dynamically expanded as required; see the discussion of the `grow_ary()` function in Section 4.5. The `sem_array` structure is allocated by a `semget()` system call and deleted by a `semctl(IPC_RMID)` system call. The individual semaphores in a set are each represented by a `sem` structure.

Each `semop()` system call presents the `semid` for the semaphore, which must be looked up in this data structure to locate the corresponding `sem_array`. Thus, each and every semaphore operation requires that this data structure be traversed.

The `ipc_ids` field `ary` is a `spinlock_t` that protects the entire data structure. This simple locking design prevents System-V semaphore operations from proceeding in parallel. In addition, the cacheline containing the `ary` spinlock is thrashed among all CPUs.

Use of RCU permits fully parallel operation of different semaphores and fully parallel translation of a semaphore ID into the corresponding `sem_array` pointer. However, a few changes to the data structure are required, as shown in Figure 9. To begin with, the `ipc_id` array and the `sem_array` are each prefixed with an `ipc_rcu_kmalloc` structure which contains the `rcu_head` structure that RCU's `call_rcu()`

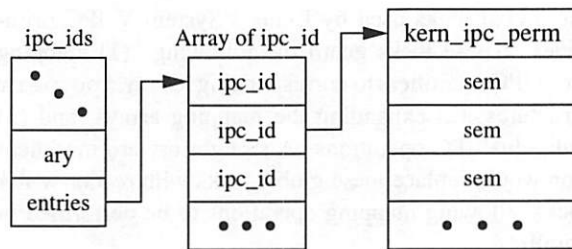


Figure 8: Semaphore Structures with Global Locking

function needs to track these structures during a grace period. In addition, since there is no longer a global ary lock, each individual `sem_array` must have its own individual lock to protect operations on the corresponding set of semaphores.

The final change is motivated by fact that the translation from semaphore ID to `kern_ipc_perm` cannot tolerate the stale data that could result when an ID translation races with an `semctl (IPC_RMID)` removing that same ID. The possibility of stale data is avoided through use of a `deleted` flag in the `kern_ipc_perm` structure, guarded by that structure's `lock` field. This `deleted` flag is set just after removing the corresponding `sem_array` but before starting the grace period. The entire removal operation is performed holding the `lock` field in the `kern_ipc_perm` structure. Any attempt to lock a semaphore structure that has the `deleted` flag set then behaves as if the structure is nonexistent, as will be shown in the following sections.

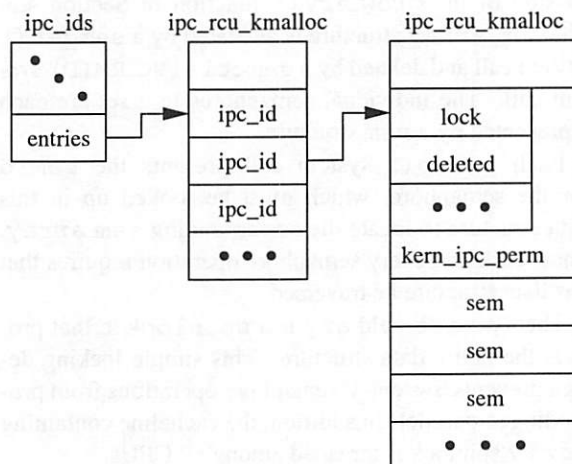


Figure 9: Semaphore Structures with RCU

## 4.2 Semaphore Removal

The deletion process is performed by `ipc_rmids`, as shown in Figure 10. This function is called with the

```

1 struct kern_ipc_perm*
2 ipc_rmids(struct ipc_ids* ids, int id)
3 {
4     struct kern_ipc_perm* p;
5     int lid = id % SEQ_MULTIPLIER;
6     if (lid >= ids->size)
7         BUG();
8
9     p = ids->entries[lid].p;
10    ids->entries[lid].p = NULL;
11    if (p == NULL)
12        BUG();
13    ids->in_use--;
14
15    if (lid == ids->max_id) {
16        do {
17            lid--;
18            if (lid == -1)
19                break;
20        } while (ids->entries[lid].p == NULL);
21        ids->max_id = lid;
22    }
23    p->deleted = 1;
24    return p;
25 }

```

Figure 10: Semaphore Deletion

lock held, and returns with it held. Lines 4-9 obtain a pointer to the `sem_array` structure. Line 10 NULLs the pointer to `sem_array`, removing any path from a permanent variable to this structure. Lines 11-12 perform a debug check, which could be triggered by locking design bugs, among other things. Lines 13-22 adjust the count of semaphores in response to the removal of this one, and then, if this semaphore had the largest ID, scans down the array of `ipc_ids` to find the new largest ID. Line 23 sets the `deleted` flag, so that the next acquisition of the lock will fail (see Section 4.3 below), and Line 24 returns a pointer to the newly removed semaphore. The semaphore's memory is freed up by a call to `ipc_rcu_free()` by `freeary()`, which is `ipc_rmids()`'s caller and which also performs other cleanup actions, including waking up any processes that were sleeping on the newly removed semaphore.

The `deleted` flag, once set, makes the corresponding semaphore set appear to be freed up even though it is still in memory awaiting expiration of its grace period, as will be shown in the next section.

## 4.3 Semaphore Lock Acquisition

As noted earlier, each `semop()` system call presents the `semid` for the semaphore, which must be looked up to locate the corresponding `sem_array`. In addition, the semaphore state must be locked. The `semop()` system call invokes the `ipc_lock()` kernel function to do this lookup and locking, and later invokes the `ipc_unlock()` kernel function to do the corresponding unlocking. Since the `ipc_lock()` kernel function was responsible for the lock contention that motivated use of RCU, we focus on `ipc_lock()` and the functions that it interacts with.



```

1 struct kern_ipc_perm*
2 ipc_lock(struct ipc_ids* ids, int id)
3 {
4     struct kern_ipc_perm* out;
5     int lid = id % SEQ_MULTIPLIER;
6     struct ipc_id* entries;
7
8     rcu_read_lock();
9     if(lid >= ids->size) {
10         rcu_read_unlock();
11         return NULL;
12     }
13     /* barrier syncs with grow_ary() */
14     smp_rmb();
15     entries = ids->entries;
16     read_barrier_depends();
17     out = entries[lid].p;
18     if(out == NULL) {
19         rcu_read_unlock();
20         return NULL;
21     }
22     spin_lock(&out->lock);
23     /* in case ipc_rmid() just freed ID */
24     if (out->deleted) {
25         spin_unlock(&out->lock);
26         rcu_read_unlock();
27         return NULL;
28     }
29     return out;
30 }

```

Figure 11: Detecting Semaphore Deletion

Since the read-code is lock-free, nothing will prevent `ipc_lock()` from racing with `ipc_rmid()`, thus possibly gaining a reference to the structure after it is marked deleted. Figure 11 shows how `ipc_lock()` handles this race. Note that the `ids` argument is a pointer to the sole permanent variable, while the `out` pointer declared in Line 4 is a temporary variable. Line 5 computes the “hash” used to access the array of `ipc_ids`. Line 8 marks the beginning of the RCU read-side critical section. In preemptive kernels, this will disable preemption; in non-preemptive kernels, it does absolutely nothing other than serve as a documentation aid. Lines 9-12 check for the specified ID being out of range, returning `NULL` for failure if so. Line 14 allows for interactions with the `grow_ary()` function on multiprocessors with weak memory consistency models as described in Section 2.4. Note that since the semaphore implementation does not use linked lists, these memory-barrier primitives must be invoked explicitly – the RCU variants of the Linux list-manipulation primitives cannot be used. Lines 15-17 obtain a stable pointer to the semaphore structure. The `read_barrier_depends()` allows for interactions with the `grow_ary()` function on multiprocessors with extremely weak memory consistency models, such as the Alpha. Lines 18-21 attempt to get a reference to the semaphore structure, returning `NULL` if there is no such structure (perhaps due to the specified ID no longer being valid). Line 22 acquires the semaphore structure’s lock. Lines 24-28 check the `deleted` flag to determine if the semaphore is being removed, and,

```

1 void ipc_rcu_free(void* ptr, int size)
2 {
3     struct ipc_rcu_kmalloc *free;
4     free = ptr - sizeof(*free);
5     call_rcu(&free->rcu,
6             (void (*)(void *))kfree,
7             free);
8 }

```

Figure 12: Freeing a Semaphore

if such a race occurred, returns `NULL` to signal failure. Note that because `ipc_lock()` does not block, the normal RCU grace period keeps the semaphore structure around for long enough that there is no danger of this structure being freed up before `ipc_lock()` can check the `deleted` flag. Finally, Line 29 returns a pointer to the semaphore structure, having successfully translated the specified ID. Note that this function returns with the semaphore’s lock held inside a RCU read-side critical section. The `ipc_unlock()` function therefore releases the semaphore’s lock and then ends the RCU read-side critical section by executing a `rcu_read_unlock()`.

## 4.4 Semaphore Deferred Deletion

Since `ipc_lock()` can gain a reference to a semaphore as it is being removed, a grace period must elapse between the removal and the actual freeing of the corresponding data structures, as illustrated by Figure 12, which shows a simplified version of `ipc_rcu_free()` function. The actual function is more complex due to the fact that blocks of memory larger than a page must be freed with `vfree()` rather than `kfree()`. Line 4 computes a pointer to the beginning of the structure (see Figure 9), which is an `ipc_rcu_kmalloc()`, which in turn is just a wrapper around an `rcu_head` structure (see Figure 4). This wrapping allows more common code between the `kmalloc()` and `vmalloc()` cases. Lines 5-7 then pass to `call_rcu()` pointers to the `rcu_head` structure, to the `kfree()` function, and to the semaphore structure. The `call_rcu()` function uses the `rcu_head` structure to queue up the semaphore structure during the grace period. The actual invocation of the `kfree()` function on the semaphore structure is deferred until after the end of a subsequent grace period.

## 4.5 Semaphore Array Expansion

If a large number of semaphores are created, the kernel will need to expand the `ipc_id` array. Use of RCU dictates that this expansion occur in parallel with ongoing searching by `ipc_lock()`. The function `grow_ary()`, shown in Figure 13, implements this expansion.

Lines 8-11 do limit checking. Lines 13-21 allocate the new array, copy the old array to the first part of the

```

1 static int grow_ary(struct ipc_ids* ids,
2                     int newsize)
3 {
4     struct ipc_id* new;
5     struct ipc_id* old;
6     int i;
7
8     if(newsize > IPCMNI)
9         newsize = IPCMNI;
10    if(newsize <= ids->size)
11        return newsize;
12
13    new = ipc_rcu_alloc(sizeof(struct ipc_id) *
14                       newsize);
15    if(new == NULL)
16        return ids->size;
17    memcpy(new, ids->entries,
18          sizeof(struct ipc_id)*ids->size);
19    for(i=ids->size; i<newsize; i++) {
20        new[i].p = NULL;
21    }
22    old = ids->entries;
23    i = ids->size;
24
25    smp_wmb();
26    ids->entries = new;
27    smp_wmb();
28    ids->size = newsize;
29
30    ipc_rcu_free(old, sizeof(struct ipc_id)*i);
31    return ids->size;
32 }

```

Figure 13: Expanding the Array of Pointers to Semaphores

new array, and initialize the remainder of the new array. Lines 22 and 23 retain the size of the old array and a pointer to it. Line 25 is a memory barrier that prevents the CPU and the compiler from reordering the array initialization with the assignment of the pointer. Any such reordering could cause other CPUs to see uninitialized segments of the array, possibly crashing (or worse!). Line 26 switches the pointer over to the new array, but any accesses at this point will recognize only the old entries as valid, since the size is still the old size. Line 27 is a memory barrier that prevents the CPU and the compiler from reordering the assignment of the size to precede the pointer assignment. If such a reordering were to occur while a user was attempting to access an erroneously larger `semid`, the kernel would run off the end of the old array, again, possibly crashing. Line 28 updates the size, so that new semaphores with larger `semids` may now be accommodated. Line 30 invokes `ipc_rcu_free()`, which frees the old structure after a full grace period has elapsed. Note that `ipc_rcu_free()` returns immediately, having used `call_rcu()` to queue the old array for a later `kfree()`. Finally, Line 31 returns the new size of the array.

Note that no `deleted` flag is needed here, since the old version of the array is kept valid throughout the grace period. Any semaphore in existence at the start of the racing access that is still in existence when the racing access completes will still be correctly referenced by the

old array. Note that the racing access must by definition complete before the grace period ends – otherwise, it is not a grace period.

This tolerance of stale data is typical of ID-to-address mappings, and of routing tables as well.

## 4.6 Semaphore Operation

This section presents a graphical demonstration of how `grow_ary()`, `ipc_rmid()`, and `ipc_lock` operate. The figures in this section are abbreviated forms of Figure 8 and Figure 9. Figure 14 shows a system with three semaphores allocated out of a maximum of eight that could be accommodated.

The results of a concurrent `grow_ary()`, `ipc_rmid()` and creation of a new semaphore are shown in Figure 15, but with the additional `ipc_id` array elements omitted from the figure. At this point, a concurrent `ipc_lock()` would see semaphore 4 as being deleted (note the "D" in the diagram), and would have no way of reaching the newly created semaphore 2. The lack of visibility to semaphore 2 is legal, since this semaphore was created *after* `ipc_lock()` started execution. A subsequent `ipc_lock()` would see semaphores 0, 2, and 6, but would not newly deleted semaphore 4.

Finally, Figure 16 shows the state of the system after a grace period. The old `ipc_id` array has been freed, as has semaphore 4. Because the grace period has completed, there can no longer be any references either to the old array or to the now-deleted semaphore 4.

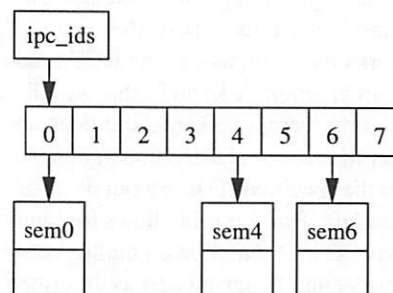


Figure 14: Semaphore Initial State

## 4.7 Semaphore Performance

Use of RCU improves the performance of System V semaphores as measured by both system-level benchmarks and focused microbenchmarks.

The Open Source Development Lab (OSDL) used a DBT1 benchmark to evaluate system-level performance, comparing Andrew Morton's Linux 2.5.42-mm2 both with and without `ipc-rcu`. These tests were run on an Intel<sup>(R)</sup> dual-CPU 900MHz PIII with 256MB of mem-

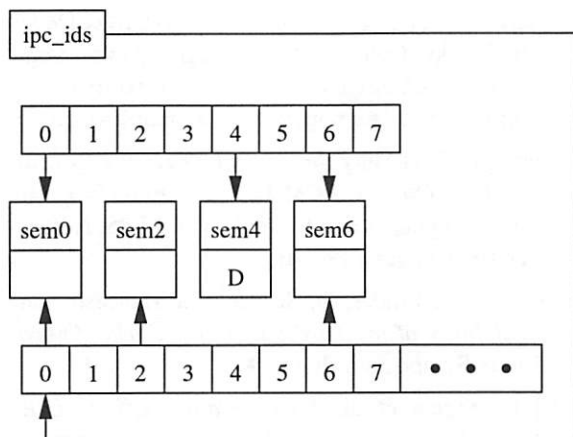


Figure 15: Semaphore Structures After Array Replacement

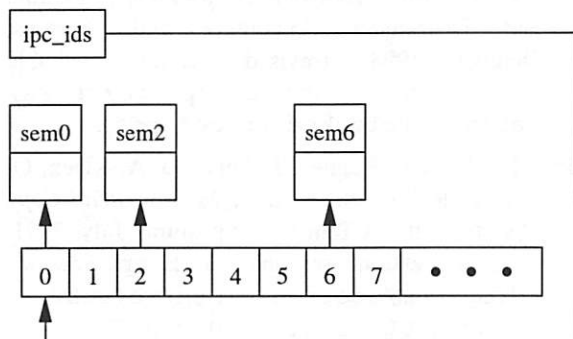


Figure 16: Semaphore Structures After Grace Period

ory.

The raw transaction rate for each of the five runs with each kernel are shown in Figure 17. The erratic results for the stock kernel are not unusual for workloads with lock contention. The reason for this is that if the lock contention is not too extreme, relatively deterministic workloads can “get lucky” such that multiple CPUs happen to be less likely to be contending for the same lock at the same time. As shown in Table 4, the difference is statistically significant: not only is *ipc-rcu*’s average three standard deviations above that of the stock kernel, but *ipc-rcu*’s smallest value of 90.4 TPS exceeds the stock kernel’s median of 87.6 TPS.

Bill Hartner [Hartner02] constructed a System V

Kernel	Average	Standard Deviation
2.5.42-mm2	85.0	7.5
2.5.42-mm2+ipc-rcu	89.8	1.0

Table 4: DBT1 Database Benchmark Results (TPS)

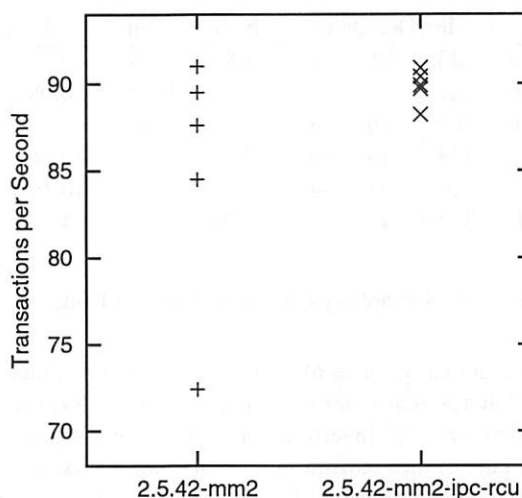


Figure 17: DBT1 Database Benchmark Raw Results

Kernel	Run 1	Run 2	Avg
2.5.42-mm2	515.1	515.4	515.3
2.5.42-mm2+ipc-rcu	46.7	46.7	46.7

Table 5: semopbench Microbenchmark Results (seconds)

semaphore microbenchmark named *semopbench* and ran it on an Intel 8-CPU 700 MHz PIII system. The results in Table 5 clearly show the order-of-magnitude reduction in runtime obtained by applying the reader-writer-locking/RCU analogy with RCU to System V IPC mechanisms.

## 4.8 Semaphore Complexity

The RCU changes to the System V IPC implementations inflicted less than 5% expansion of code size, as shown in Table 6. This change increased the overall code size by only 151 lines. The RCU implementation itself (which is also used by both module unloading and the IP route cache) adds only an additional 408 lines of code. This order-of-magnitude performance benefit is well worth the modest increase in complexity.

Of course, the system-level performance increase is a much smaller 5.3%. On the other hand, the 151-line increase in code size is an insignificant fraction of the 11.7 million lines of code in the full kernel, and even this does not include the size of the database and other software involved in the benchmark.

## 5 Conclusions and Future Plans

Since the modified version of *rcu-poll* was unable to match the performance of *rcu-ltimer*, and since there are

	Ins/Del/Delta			Total Lines		% Delta
				New	Old	
msg.c	23	26	-3	885	888	-0.34%
sem.c	29	30	-1	1289	1290	-0.08%
shm.c	102	69	33	785	752	4.39%
util.c	178	13	165	581	416	39.66%
util.h	10	53	-43	64	107	-40.19%
Total	342	191	151	3604	3453	4.37%

Table 6: Semaphore Change in Lines of Code

currently no known uses of RCU that require low grace-period latency, *rcu-ltimer* is used in the Linux 2.5 kernel. If needed, we will investigate use of *rcu-poll*'s grace-period-latency mechanisms in *rcu-ltimer* after a fixed delay. A modified version of *rcu-sched* may attain even lower overheads than those of *rcu-ltimer*.

We will continue investigating how RCU may be used in the Linux kernel, including using it to provide NMI handler support and applying it to the `tasklist_lock` to eliminate some starvation scenarios that have been observed in the Linux 2.5 kernel. Experience gained from use of RCU in DYNIX/ptx<sup>(R)</sup>, K42 [Gamsa99], and Linux indicate that it will continue to be quite helpful in obtaining dramatic performance improvements with little increase in complexity.

## 6 Acknowledgments

We owe thanks to Bill Hartner and Cliff White for running many benchmarks, to Rusty Russell, Anton Blanchard, Paul Mackerras, and the rest of the IBM OzLabs group for many valuable discussions, to Andrew Morton, Hugh Dickens, Davide Libenzi, and Rusty Russell again for their careful review of much code, and to Matt Dobson, the anonymous referees, and shepherd Ray Bryant for their substantial help in rendering this paper human-readable. We are indebted to Dan Frye, Hans Tannenber, Chris Maher, Randy Kalmata, Vijay Sukthankar, and Hugh Blemings for their support of this effort.

## References

- [Cao02] M. Cao [PATCH] Latest IPC lock patch-2.5.44, Linux Kernel Mailing List, October 2002. <http://marc.theaimsgroup.com/?l=linux-kernel&m=103610704923787&w=2>.
- [Gamsa99] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. *Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system*, Proceedings of the 3rd Symposium on Operating System Design and Implementation, New Orleans, LA, February, 1999.
- [Hartner02] B. Hartner. *semopbench 1.0.0*, IBM DeveloperWorks, October 2002. <http://www.ibm.com/developerworks/opensource/linuxperf/semopbench/semopbench.c>.
- [Herlihy93] M. Herlihy. *Implementing Highly Concurrent Data Objects*, ACM Transactions on Programming Languages and Systems, vol. 15 #5, November 1993, pages 745-770.
- [Linder02a] H. Linder, D. Sarma, and Maneesh Soni. *Scalability of the Directory Entry Cache*, Ottawa Linux Symposium, June 2002.
- [LSE] D. Sarma et al. *Linux Scaling Effort (LSE)*, SourceForge Project, April 2002. <http://prdownloads.sourceforge.net/lse/>.
- [McK98a] P. E. McKenney and J. D. Slingwine. *Read-copy update: using execution history to solve concurrency problems*, Parallel and Distributed Computing and Systems, October 1998. (revised version available at <http://www.rdrop.com/users/paulmck/rclockpdcproof.pdf>).
- [McK01b] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, M. Soni. *Read-Copy Update*, Ottawa Linux Symposium, July 2001. (revised version available at [http://www.rdrop.com/users/paulmck/rclock/rclock\\_OLS.2001.05.01c.pdf](http://www.rdrop.com/users/paulmck/rclock/rclock_OLS.2001.05.01c.pdf)).
- [McK02a] P. E. McKenney, D. Sarma, A. Arcangel, A. Kleen, O. Krieger, and R. Russell. *Read-Copy Update*, Ottawa Linux Symposium, July 2002. (revised version available at <http://www.rdrop.com/users/paulmck/rclock/rcu.2002.07.08.pdf>).
- [Michael02a] M. M. Michael. *Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes*, In Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing, pages 21-30, July 2002.
- [Michael02b] M. M. Michael. *High Performance Dynamic Lock-Free Hash Tables and List-Based Sets*, In Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architecture, pages 73,82, August 2002.
- [Pugh90] W. Pugh. *Concurrent Maintenance of Skip Lists*, Department of Computer Science, University of Maryland, CS-TR-2222.1, June 1990.
- [Sarma02] D. Sarma [PATCH] Read-Copy Update 2.5.42, Linux-Kernel Mailing List, October 2002. <http://marc.theaimsgroup.com/?l=linux-kernel&m=103461974415359&w=2>.



[Spraul01] M. Spraul *Re: RFC: patch to allow lock-free traversal of lists with insertion*, Linux-Kernel Mailing List, October 2001. <http://marc.theaimsgroup.com/?l=linux-kernel&m=100264675012867&w=2>.



# An Implementation of User-level Restartable Atomic Sequences on the NetBSD Operating System

Gregory McGarry

g.mcgarry@ieee.org

## Abstract

This paper outlines an implementation of restartable atomic sequences on the NetBSD operating system as a mechanism for implementing atomic operations in a mutual-exclusion facility on uniprocessor systems. Kernel-level and user-level interfaces are discussed along with implementation details. Issues associated with protecting restartable atomic sequences from violation are considered. The performance of restartable atomic sequences is demonstrated to outperform syscall-based and emulation-based atomic operations. Performance comparisons with memory-interlocked instructions demonstrate that restartable atomic sequences continue to provide performance advantages on modern hardware. Restartable atomic sequences are now the preferred mechanism for atomic operations in the NetBSD threads library on all uniprocessor systems.

## 1 Introduction

The NetBSD Project is currently adopting a new threads system based on scheduler activations[6]. Part of this project is the implementation of a POSIX-compliant threads library that utilises the scheduler activations interface. The motivation for the threads project is to support the multi-threaded programming model which is becoming increasingly popular for application development. Multi-threaded applications use multiple threads to aid portability to multiprocessor systems, and as a way to manage server concurrency even when no true system parallelism is available. To support multi-threaded applications, the POSIX standard specifies a mutual-exclusion facility to serialise access and guarantee consistency of shared data. Even on uniprocessor systems, mutual-exclusion facilities are necessary to protect shared data against an interleaved thread schedule. Interleaving can occur when a thread blocks on a resource or when a thread is preempted, causing another thread to assume control of the processor.

The scheduler activations threading model also places additional demand on the mutual exclusion facility. The primary advantage of scheduler activations is that it combines the simplicity of a kernel-based threading system and the performance of a user-level threading

system[1]. While the kernel component of the model controls the switching of execution contexts, the user-level component is responsible for scheduling threads onto the available execution contexts. The user-level component contains a complete scheduler implementation with shared scheduling data which must be protected by the mutual-exclusion facility. Consequently, the mutual-exclusion facility is a critical component of the scheduler activations threading model.

The basic building block for any mutual-exclusion facility, whether it be a *blocking* or *busy-wait* construct, is the *fetch-modify-write* operation[5]. The fetch-modify-write operation reads a boolean flag that indicates ownership of shared data. The operation modifies the flag from false to true, thereby acquiring ownership. The fetch-modify-write operation must execute atomically (without interruption) to ensure the flag state is maintained consistent across all contending threads.

Modern systems generally provide sophisticated processor primitives within the hardware in the form of memory-interlocked instructions and bus support to ensure that a given memory location can be read, modified and written atomically. The specific primitive varies between processors, however common instructions include *test-and-set*, *fetch-and-store* (swap), *fetch-and-add*, *load-locked/store-conditional* and *compare-and-swap*[5, 2]. Unfortunately, there are two problems associated with the use of memory-interlocked instructions within a user-level threads library:

- Memory-interlocked instructions incur overhead since the cycle time for an interlocked memory access is several times greater than that for a non-interlocked access. The overhead associated with memory-interlocked instructions is due to memory and interconnection contention.
- Not all processors support memory-interlocked instructions and therefore cannot provide atomic operations for a mutual-exclusion facility. Example processors include the MIPS R3000, VR4100 and ARM2 processors. Interestingly, processor manufacturers are choosing to introduce new processors without memory-interlocked instructions. These processors generally provide a subset of the complete processor specification and primarily target

embedded or low-power applications.

Both of these problems are important to the NetBSD Project due to the large number of hardware systems that are supported.

On multiprocessor systems, the hardware is expected to provide the necessary atomic operations. Multiprocessor atomic operations for synchronisation have been extensively investigated by Mellor-Crummey and Scott[5]. However, the majority of systems supported by the NetBSD operating system are uniprocessor systems. Therefore, it is desirable to find an alternate mechanism for atomic operations which works efficiently on all uniprocessor systems.

The simplest mechanism for implementing an atomic operation is to request the kernel to perform the operation of behalf of the user-level thread. A call to a specific *system call* can be used to enter the kernel. While inside the kernel, the scheduler can be paused so that the thread is guaranteed not to be preempted while a non-atomic fetch-modify-write operation is performed. Alternatively, an invalid instruction can be executed by the thread to cause an exception which can be intercepted by the kernel to *emulate* an atomic fetch-modify-write operation. An advantage of instruction emulation is that the invalid instruction can be forward compatible with newer versions of the processor which do support explicit atomic operations. However, both syscall-based and emulation-based solutions incur significant overhead by entering the kernel.

Atomic operations can also be implemented using a software reservation mechanism. With the software reservation mechanism, a thread must register its intent to perform an atomic operation and then wait until no other thread has registered a similar intent before proceeding. The most widely recognised algorithm based on software reservation is Lamport's mutual-exclusion algorithm[3].

In Lamport's algorithm, the atomic operation is protected by two variables; one to indicate ownership of the atomic operation and another to place reservations. Each thread registers in a global array its intent to perform an atomic operation. The thread then places a reservation into the reservation variable before testing the ownership variable. If a thread determines that ownership is held by another thread, there is *contention*, and the thread must wait until ownership is relinquished. When the thread determines that ownership has been relinquished, it assigns its ownership to the atomic operation then checks that it still holds the reservation. If another thread holds the reservation then a *collision* has occurred. The thread then waits for all contending threads to acquire ownership and remove their intent from the global array. The thread then restarts the procedure from the beginning.

The software reservation mechanism works equally

well on both uniprocessor and multiprocessor systems. However, even for the case when no contention and no collisions occur, reservation-based algorithms require several memory accesses per atomic operation. Additionally, these algorithms have storage requirements that increase linearly with the number of potential contending threads. For a user-level threads library, it is not always possible to know the maximum number of threads likely to be used in an application. For these reasons, the software reservation mechanism was not considered further.

The mechanism of restartable atomic sequences has been proposed to address the problems outlined above for implementing atomic operations on uniprocessor systems[2]. The basic concept of restartable atomic sequences is that a user-level thread which is preempted within a restartable atomic sequence is resumed by the kernel at the beginning of the atomic sequence rather than at the point of preemption. This guarantees that the thread eventually executes the instruction sequence atomically.

This paper outlines the implementation of restartable atomic sequences to provide the atomic operations required by the POSIX-compliant threads library on the NetBSD operating system. Section 2 discusses the concept of restartable atomic sequences. Section 3 presents details of the implementation on the NetBSD operating system. Section 4 compares the performance of restartable atomic sequences with other mechanisms for implementing atomic operations including memory-interlocked instructions, instruction emulation and a syscall-based mechanism. Section 4 also examines the cost associated with restartable atomic sequences. Section 5 discusses the application of restartable atomic sequences in the POSIX-compliant threads library and presents some benchmarks of the mutual-exclusion facility in the threads library. Finally, Section 6 presents conclusions.

## 2 Restartable Atomic Sequences

Restartable atomic sequences is a mechanism to implement atomic operations on uniprocessor systems. Responsibility for executing the instruction sequence atomically is shared between the user-level thread and the kernel. When a thread is preempted within a restartable atomic sequence, it is resumed by the kernel at the beginning of the atomic sequence rather than at the point of preemption.

Most mechanisms used to implement atomic operations on uniprocessor systems can be called pessimistic. Their design assumes that atomicity may be violated at any moment and guards against this potential violation for every atomic operation. Restartable atomic sequences use an optimistic mechanism that assumes that



atomic sequences are rarely preempted and are inexpensive for this common case. Only when an atomic sequence is not executed atomically is it necessary to perform a recovery action to ensure atomicity.

Restartable atomic sequences require kernel support to ensure that a preempted thread is resumed at the beginning of the sequence. An application registers the address range of the atomic sequence with the kernel. If a user-level thread is preempted within a registered atomic sequence, then its execution is rolled-back to the start of the sequence by resetting the program counter saved in its process control block.

Consider the test-and-set atomic operation in Figure 1 which reads the memory address, writes to the memory address and returns the read value. The memory address could be a flag in a mutual-exclusion operation. If the test-and-set operation executes atomically, two conditions can occur. If `*addr` is unset, then it is set and the function returns the unset value. If `*addr` is set, then it remains unchanged and the function returns the set value.

Now consider what will happen if the test-and-set operation is preempted between the memory read and write operations and a collision occurs when accessing `*addr`. Again, two conditions occur. If `*addr` is unset, `old` is unset. The thread is then preempted. At this point the new thread will read `*addr` as unset, and set `*addr`. When the original thread resumes, its recorded value of `old` no longer reflects the true value of `*addr` and will also set `*addr`. Both threads will have the test-and-set operation return success to setting the memory address. If this condition occurred while the test-and-set operation was being used in a mutual-exclusion operation, then both threads would assume ownership of a shared resource.

The other collision condition occurs if `*addr` is read as set before the thread is preempted. The new thread clears `*addr`. When the original thread resumes, its recorded value of `old` no longer reflects the true value of `*addr`, it will set `*addr` but return the unset value. In this condition, the original thread believes that the memory address was already set, while the second thread has cleared the memory address. If this condition occurred while the test-and-set operation was being used in a mutual-exclusion operation, then neither thread would assume ownership of the shared resource. Neither thread would be able to reacquire ownership of the resource and *deadlock* would occur.

Atomicity of the test-and-set operation is assured by making the adjacent memory accesses between `__ras_start` and `__ras_end` a restartable atomic sequence. Now consider what will happen if the test-and-set operation is preempted between the memory read and write operations. Again, two conditions occur. If

```
int test_and_set(int *addr)
{
    int old;

    __asm __volatile("__ras_start:");
    old = *addr;
    *addr = 1;
    __asm __volatile("__ras_end:");

    return (old);
}
```

Figure 1: Implementation of a test-and-set atomic operation protected as a restartable atomic sequence.

`*addr` is unset, `old` is unset. The thread is then preempted. At this point the new thread will read `*addr` as unset, and set `*addr`. When the original thread resumes, its execution is rolled-back to `__ras_start` which corresponds to the instruction to read `*addr`. The value of `old` now corresponds to the correct value of `*addr` set by the other thread. The value of `*addr` remains unchanged and the correct value is returned by the test-and-set operation. The other collision condition occurs analogously.

Restartable atomic sequences should adhere to the following requirements:

1. have a single entry point;
2. have a single exit point;
3. restrict modifications to global/shared data;
4. not execute emulated instructions or invoke system calls; and
5. not invoke any functions.

The first requirement is to ensure that the roll-back of the program counter is valid. Nevertheless, the kernel cannot guarantee that the sequence is successfully restartable; it assumes that the application knows what it is doing. The second and third requirements are linked. Access to global/shared data should be restricted to ensure that the restartable atomic sequence is *idempotent*. An operation is idempotent if it achieves the same result irrespective of the number of times it executes. Restricting the restartable atomic sequence to a single global write in the last instruction of the restartable atomic sequence ensures that the operation is idempotent. Accordingly, a single exit point is desirable. However, if the atomic operation chooses not to modify any global data, the restartable atomic sequences may be exited at any point.

The fourth requirement is to ensure that the kernel is not entered and thus providing an opportunity for the

thread to block on a resource and be preempted. In this case, the kernel will always roll-back execution to the beginning of the restartable atomic sequence whenever the thread is unblocked, and the thread will never exit the restartable atomic sequence. The fifth requirement is to ensure that the program counter remains within the range of the registered atomic sequence.

There are two run-time costs associated with restartable atomic sequences. Because the kernel identifies restartable atomic sequences by an address range, restartable atomic sequences cannot be inlined. The inability to inline atomic sequences slightly increases the overhead of atomic operations due to the cost of subroutine calls. The second run-time cost comes from checking the program counter at each context switch. Although this test can add several cycles to the kernel's context switch path, many applications use many more atomic operations than the number of context switches, making the additional scheduling overhead negligible.

### 3 Implementation

The NetBSD operating system is well-known for its portability and support for many architectures. The portability of the operating systems stems from a clear separation of machine-dependent and machine-independent subsystems. An implementation of restartable atomic sequences clearly requires access to machine-dependent information such as the thread program counter. However, much of the implementation can be shared between all architectures and is machine-independent. Additionally, the user-level interface should be uniform across all architectures. Consequently, the primary objective of this implementation was to provide a generic interface to be utilised by all supported architectures. To that end, the implementation consists of a simple system-call interface and a largely machine-independent kernel implementation.

Initially, the implementation was intended to support atomic operations only for use by the threads library. However, it was recognised that a generic user-level interface would allow applications to find new and innovative uses of restartable atomic sequences. For example, benchmark utilities, performance counters, and profiling tools may make use of restartable atomic sequences to ensure that an analysed instruction sequence is executed without interruption. Supporting new and innovative uses seemed like a worthwhile goal.

The initial design decision was that only thread-synchronous events will cause an atomic sequence to be restarted and asynchronous events, such as interrupts, do not cause a restart. Therefore, *a restartable atomic sequence will only be restarted if the thread execution context is switched from the processor*. Asynchronous events are difficult to protect, since they must

be identified outside the context of the executing thread. Additionally, the handling of asynchronous events is a machine-dependent operation and would require invasive changes to the machine-dependent kernel. For example, on architectures such as i386 and m68k, it is difficult to provide a central location to check if the program counter is within a restartable atomic sequence since interrupts are dispatched via an interrupt table.

Another important consideration is how registered restartable atomic sequences are handled by the *fork* and *exec* system calls. Restartable atomic sequences are inherited from the parent by the child during the *fork* system call. This allows restartable atomic sequences to continue to work on children of the parent process that registered the sequences. Restartable atomic sequences are removed during the *exec* system call. This property is intuitive given that the program text has changed and the instruction sequence for a registered atomic sequence is different.

The implementation supports the registration of multiple restartable atomic sequence for a process. A list of address ranges is maintained for each process. A per-process limit of the maximum number of registered restartable atomic sequences is imposed to limit resource exhaustion.

#### 3.1 Kernel interface

All atomic sequences for a process are manipulated by the *rasctl* system call. Its prototype can be found in `<sys/ras.h>` and is implemented within the standard C library. It has a prototype given by

```
int
rasctl(void *addr, size_t len, int op)
```

The prototype is intended to be similar to the *mmap* system call, since both system calls affect the process address space. If a restartable atomic sequence is registered and the process is preempted within the range *addr* and *addr+len*, then the process is resumed at *addr*. The operations that can be applied to a restartable atomic sequence are specified by the *op* argument. Possible operations are:

- **RAS\_INSTALL**: register a new atomic sequence;
- **RAS\_PURGE**: remove a registered atomic sequence for this process; and
- **RAS\_PURGE\_ALL**: remove all registered atomic sequences for this process.

The operation affects the restartable atomic sequence immediately.

The purge operation should be considered to have undefined behaviour if there are any other runnable threads in the process which might be executing within the registered atomic sequence at the time of the purge. The

---

```

#include <sys/ras.h>

extern void __ras_start(void);
extern void __ras_end(void);

...
if (rasctl((void *)__ras_start,
          (size_t)(__ras_end - __ras_start),
          RAS_INSTALL))
    errx(1, "rasctl failed");

```

---

Figure 2: The registration process for the restartable atomic sequence presented in Figure 1.

application must be responsible for ensuring that there is some form of coordination between threads.

The *rasctl* system call will fail with *EINVAL* if an invalid operation is specified, if *addr* or *addr+len* are invalid user addresses, or if the maximum number of restartable atomic sequences per process is exceeded. It will return *ESRCH* if the restartable atomic sequence cannot be found during a purge operation.

Figure 2 shows an example of registering the restartable atomic sequence for the test-and-set atomic operation presented in Figure 1.

### 3.2 Kernel implementation

All registered restartable atomic sequences for a process are recorded in a linked list, *p\_raslist*, located in *struct proc* of the process. Each element in the linked list records the start address and end address of a single registered restartable atomic sequence. Additionally, a counter is available to record the number of restarts actioned for the restartable atomic sequence. This counter can provide some interesting information but is rarely useful for most applications. Currently there is not a user-level interface to access the counter.

A counter, *p\_nras* in *struct proc* records the number of registered atomic sequences and is used to simplify the program-counter check. The program counter is checked within the *cpu\_switch()* function. The *cpu\_switch()* function is a machine-dependent function which is responsible for switching the context of the active thread on the processor. The *cpu\_switch()* function has a pointer to the *proc* structure passed as the first argument, and a check if *p\_nras* is non-zero is an inexpensive test. The machine-dependent implementation code on the i386 adds merely three instructions to the main execution path for the case of no registered atomic sequences. If *p\_nras* is non-zero, then *ras\_lookup()* is invoked to compare the program counter with all registered

restartable atomic sequences. The *ras\_lookup()* function is machine-independent and has the function prototype

```

caddr_t
ras_lookup(struct proc *p,
           caddr_t addr)

```

It searches the registered restartable atomic sequences for process *p* which contains the user address *addr*. If the address *addr* is found within a restartable atomic sequence, then the restart address of the restartable atomic sequence is returned, otherwise -1 is returned. In the case of a match, the machine-dependent code in *cpu\_switch()* uses the start address to reset the program counter in the process control block.

The *RAS\_INSTALL* and *RAS\_PURGE* operations of the *rasctl* system call invoke the *ras\_install()* and *ras\_purge()* functions. They have the prototypes

```

int
ras_install(struct proc *p,
            caddr_t addr, size_t len)
int
ras_purge(struct proc *p,
           caddr_t addr, size_t len)

```

The *ras\_install()* function will return *EINVAL* if *addr* or *addr+len* are invalid user addresses, or if the maximum number of restartable atomic sequences per process is exceeded. The *ras\_purge()* function will return *ESRCH* if the specified restartable atomic sequence has not been registered.

The *ras\_fork()* function is used to copy all registered restartable atomic sequences for a process to another. It is primarily during the *fork* system call when the sequences are inherited from the parent by the child. It has the prototype

```

int
ras_fork(struct proc *p1,
         struct proc *p2)

```

The *ras\_purgeall()* function is used to remove all registered restartable atomic sequences for a process. It is primarily used to remove all registered restartable atomic sequences for a process during the *exec* system call and to perform the *RAS\_PURGE\_ALL* operation for the *rasctl* system call. It has the prototype

```

int
ras_purgeall(struct proc *p)

```

The *ras\_fork()* and *ras\_purgeall()* functions are guaranteed to complete successfully.



### 3.3 Additional kernel issues

Restartable atomic sequences are user-level instruction sequences that receive special consideration by the kernel. In addition to checking if the program counter is within a restartable atomic sequence when a thread context is restored, it is also important for the kernel not to violate the prerequisites for their correct operation. One potential problem for the kernel is the *ptrace* facility.

The *ptrace* facility provides tracing and debugging facilities. It allows a process (the tracing process) to control another (the traced process). Most of the time, the traced process runs normally, however the *ptrace* facility provides some kernel capabilities to modify the behaviour of the traced process. If the traced process contains registered restartable atomic sequences then the kernel must ensure that they are protected from modification by the tracing process, otherwise one or both of the processes will fail to perform as expected.

There are two specific cases which must be considered:

- The tracing process attempts to write to a restartable atomic sequence (PT\_WRITE\_I). An example is when the tracing process attempts to set a breakpoint.
- The traced process attempts to single-step into a restartable atomic sequence (PT\_STEP).

The first case is handled within the machine-independent *ptrace* facility. Each write to the code segment is first checked to ensure that the write is not to a restartable atomic sequence. Attempting to write to a restartable atomic sequence fails.

The second case is more difficult to handle, since different architectures handle single-stepping mode differently. For example, the MIPS R3000 processor does not have a single-stepping or tracing mode and the facility is generally handled through software emulation. Software emulation works by replacing the next instruction with an invalid opcode which generates an exception that the kernel identifies and handles specifically. Protecting the restartable atomic sequences during emulation is the same as for the first case discussed above. However, care must be taken, since the same emulation technique is used to single-step the kernel inside the kernel debugger, and restartable atomic sequences are not supported within the kernel.

Other architectures such as i386 and m68k provide tracing support in hardware. On these architectures the trace trap must check if the program counter is within a restartable atomic sequence before dispatching the event to the tracing process via the *ptrace* facility. The usual procedure is to continue stepping through the restartable atomic sequence and only dispatch the event on the first instruction after the atomic sequence.

## 4 Performance Evaluation

In this section, the performance of restartable atomic sequences is compared with the competing mechanisms. The test-and-set atomic operation based on restartable atomic sequences is compared with a syscall-based mechanism, instruction emulation on the MIPS R3000 processor and memory-interlocked instructions. The overhead associated with checking the program counter during a context switch is also investigated.

All microbenchmarks presented in this section are based on mutual exclusion mechanisms with a test that enters a critical section using a *test-and-set* lock and leaves the critical section by clearing the *test-and-set* lock. The benchmark uses a single thread, so that no contention occurs. The benchmark is measuring the performance of the basic processor architecture, memory system and mutual exclusion mechanism. The measures are determined by executing the benchmark in a loop one million times. The loop overhead and overhead for clearing the lock is eliminated in the published measures.

As already mentioned, restartable atomic sequences use an optimistic mechanism that assumes that atomic sequences are rarely preempted and are inexpensive for this common case. By way of example, a system with a 100MHz i486DX processor executes one million iterations of the benchmark outlined above in 0.38 seconds with only 4 restarts actioned.

### 4.1 Syscall-based atomic operations

The syscall-based mechanism uses the kernel to perform all the necessary actions to ensure atomicity. The mechanism only works on uniprocessor systems and is successful because the NetBSD kernel is not preemptable[4]. Support is provided by two system calls to acquire and release the lock on behalf of the thread. These system calls were added to a NetBSD kernel for the explicit purpose of comparing its performance with restartable atomic sequences.

The system calls take the address in the process address of the lock. The *acquire* system call will block the current thread until the lock is released. The *release* system call will release the lock to any blocked threads. The system calls are implemented using the `tsleep()`/`wakeup()` kernel facility.

The elapsed times to execute the test-and-set atomic operation based on the syscall mechanism and restartable atomic sequences for various processors are shown in Table 1. From this benchmark it can be seen that on the MIPS R3000 processor, restartable atomic sequences provide almost ninety-fold performance improvement over syscall-based atomic operations. The run-time cost for syscall-based atomic operations is high. The kernel must be invoked on every atomic operation, requiring that a trap be fielded, dispatched and ar-



system	RAS	syscall
DECstation 5000/25	0.40	35.6
100MHz i486DX	0.32	21.5
DECstation 5000/260	0.17	9.7
Alchemy Pb1000 EVB	0.04	2.4
Broadcom BCM91250A EVB	0.04	1.4

Table 1: Elapsed times (in microseconds) to execute the test-and-set atomic operation based on the syscall mechanism and restartable atomic sequences (RAS).

guments checked. Modern processors in the MIPS family appear to be less sensitive to system-call overhead, however restartable atomic sequences continue to provide a significant performance improvement.

## 4.2 Instruction emulation

The original MIPS instruction set architecture (ISA) implemented in the R3000 processor did not provide any memory-interlocked instructions. The MIPS II ISA implemented in the R4000 and most subsequent processors introduced a load-locked/store-conditional instruction pair. The MIPS R3000 processor will generate a trap if it attempts to execute a load-locked or store-conditional instruction. This trap can be intercepted by the kernel and the necessary actions performed to adequately emulate the instruction pair.

Instruction emulation has the advantage that software can be written for any processor in the family and the unsupported instructions are supported transparently.

The NetBSD kernel currently emulates the load-locked and store-conditional instructions for the MIPS R3000 processor. The current implementation only operates on uniprocessor systems and is successful because the kernel is not preemptable[4].

Although instruction emulation requires no special hardware, its run-time cost is high. Similar to the syscall-based mechanism, the kernel must be invoked on every emulated instruction, requiring that a trap be fielded, dispatched and arguments checked. A single test-and-set atomic operation on the DECstation 5000/25 based on restartable atomic sequences can execute in 0.4 microseconds. The test-and-set atomic operation using instruction emulation executes in 56 microseconds. The cost of instruction emulation is higher than the syscall-based test-and-set atomic operation, since each test-and-set operation uses a load-locked and store-conditional instruction, which generates two traps to the kernel to emulate the instructions. Therefore, on the MIPS R3000 processor, the syscall-based mechanism is faster than instruction emulation.

## 4.3 Memory-interlocked instructions

The performance of atomic operations based on restartable atomic sequences is compared with memory-interlocked instructions on processors that provide such functionality. Two memory-interlocked implementations are considered. An *inlined* version uses compiler and assembler optimisations to schedule the memory-interlocked instructions in the instruction stream at the point of invocation. A *non-inlined* version wraps the memory-interlocked instructions in a function call. Due to the overhead associated with dispatching a function call, the inlined version is almost always expected to provide a performance gain. As mentioned in Section 2, restartable atomic sequences cannot be inlined.

To compare the performance of the two implementations of memory-interlocked instructions and restartable atomic sequences, a performance index is introduced. The metric is calculated by

$$M = 100 \left( \frac{t_{NI} - t}{t_{NI}} \right),$$

where  $t$  is the execution time of the atomic-operation mechanism and  $t_{NI}$  is the execution time for a non-inlined memory-interlocked instruction. Therefore, the performance index provides an indication of improvement over a non-inlined memory-interlocked instruction. A performance index of zero indicates no performance improvement. A performance index of one-hundred indicates zero cost associated with an operation, or effectively infinite performance improvement.

The performance indices comparing atomic operations based on restartable atomic sequences and inlined memory-interlocked instructions are shown in Table 2. The table is ordered approximately corresponding to the processing power (or age) of the processor.

Table 2 shows that for older processors, restartable atomic sequences exhibit an additional cost over memory-interlocked instructions. On these machines, restartable atomic sequences are paying the penalty of the function call. The modern processors tend to indicate a potential performance improvement of restartable atomic sequences over memory-interlocked instructions. This improvement is mainly attributed to the introduction of on-chip caches and memory controllers.

Table 2 clearly shows that the memory subsystem is crucial for efficient performance of lock operations. The Chalice CATS, Digital DNARD and Netwinder systems which have an ARM processor show significant performance improvement with restartable atomic sequences. This processor appears to be very sensitive to memory performance. However, newer processors in the ARM family such as the Xscale show the memory subsystem has been significantly improved so that inlined memory-interlocked instructions outperform restartable atomic

sequences. For the Xscale case, restartable atomic sequences are paying the penalty of being non-inlined. On this particular Xscale system, the memory controller is integrated into the CPU, the memory that memory-interlocked instruction is manipulating is cacheable, and so the memory-interlocked instruction is inexpensive.

The systems based on the Alpha processor also show significant performance improvement with restartable atomic sequences. These systems generally have small performance indices for the inlined memory-interlocked instructions; an indication that the cost associated with function invocation is negligible. The AlphaServer 1200 is a multiprocessor system, so the hardware causes significant extra bus activity. However, the benchmark was run with a uniprocessor kernel.

The microbenchmark results presented in Table 2 demonstrate that an architecture and memory system cannot necessarily provide efficient functionality compared with a combination of kernel and compiler optimisation. On modern processors, restartable atomic sequences can provide improved performance in atomic operations.

#### 4.4 Run-time overhead of restartable atomic sequences

As mentioned in Section 2, there is a run-time overhead associated with checking if the program counter is within a restartable atomic sequence on every context switch. The context-switch time was measured to obtain an indication of the cost of checking the program counter. The context-switch time was determined by measuring the time it takes a token to be passed through a pipe between two processes. The token is passed between the two processes twenty times for a single measurement. Of 16000 measurements, the shortest time was chosen as the true context-switch measurement, since it most-likely represents the uninterrupted time to perform the context switch. Context-switch times are measured for increasing number of registered atomic sequences and are shown for the DECstation 5000/25 and 100MHz i486DX in Figure 3.

Since the restartable atomic sequences are recorded in a linked list for each process the context-switch times increase linearly with increasing number of registered atomic sequences. By one-hundred registered atomic sequences the context-switch time for the DECstation 5000/25 has doubled. About 130 registered atomic sequences are required to double the context-switch time on the 100MHz i486DX.

For a small number of sequences in the list, the performance is not likely to be a significant issue. Nevertheless, one way to improve the performance might be to order the list according to the start address. Then the `ras_lookup()` function could quickly abort the

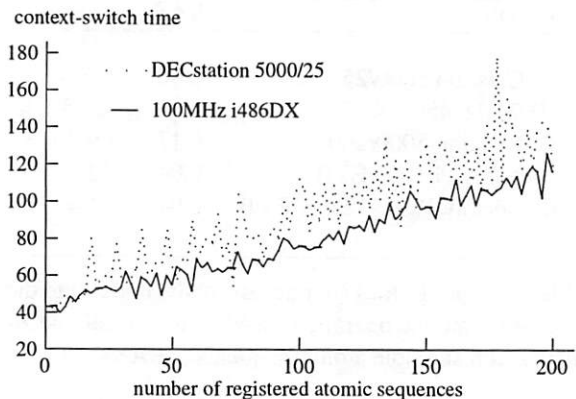


Figure 3: Context-switch time (milliseconds) for DECstation 5000/25 and 100MHz i486DX with increasing number of registered atomic sequences.

search when it finds a sequence with a start address higher than the program counter. Similarly, the first and last addresses of all restartable atomic sequences could be recorded in a header which would allow the `ras_lookup()` function to quickly check if it is necessary to traverse the list. Realistically, applications will not make use of so many restartable atomic sequences. Indeed, the current implementation places an arbitrary restriction of sixteen sequences per process. Since the threads library is currently the only user of restartable atomic sequences, only one restartable atomic sequence is expected to be registered for an application.

## 5 Discussion

Based on the performance results presented in Section 4, restartable atomic sequences is the default mechanism for implementing atomic operations in the threads library on the NetBSD operating system. The mechanism provides the foundation for the implementation of a POSIX-compliant mutual-exclusion facility and the primitives for mutual exclusion in the library scheduler.

The threads library uses a *blocking* construct in the mutual-exclusion facility. This facility provides the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. The implementation uses a test-and-set atomic operation to test the ownership of a mutual-exclusion flag (mutex). If a mutex is owned by another thread, the scheduler blocks the current thread and switches another thread onto the available execution context. Eventually a thread will release ownership of the mutex and any blocked threads waiting on the mutex will be given the execution context.

The blocking construct within the threads library uses a single test-and-set atomic operation and therefore uses

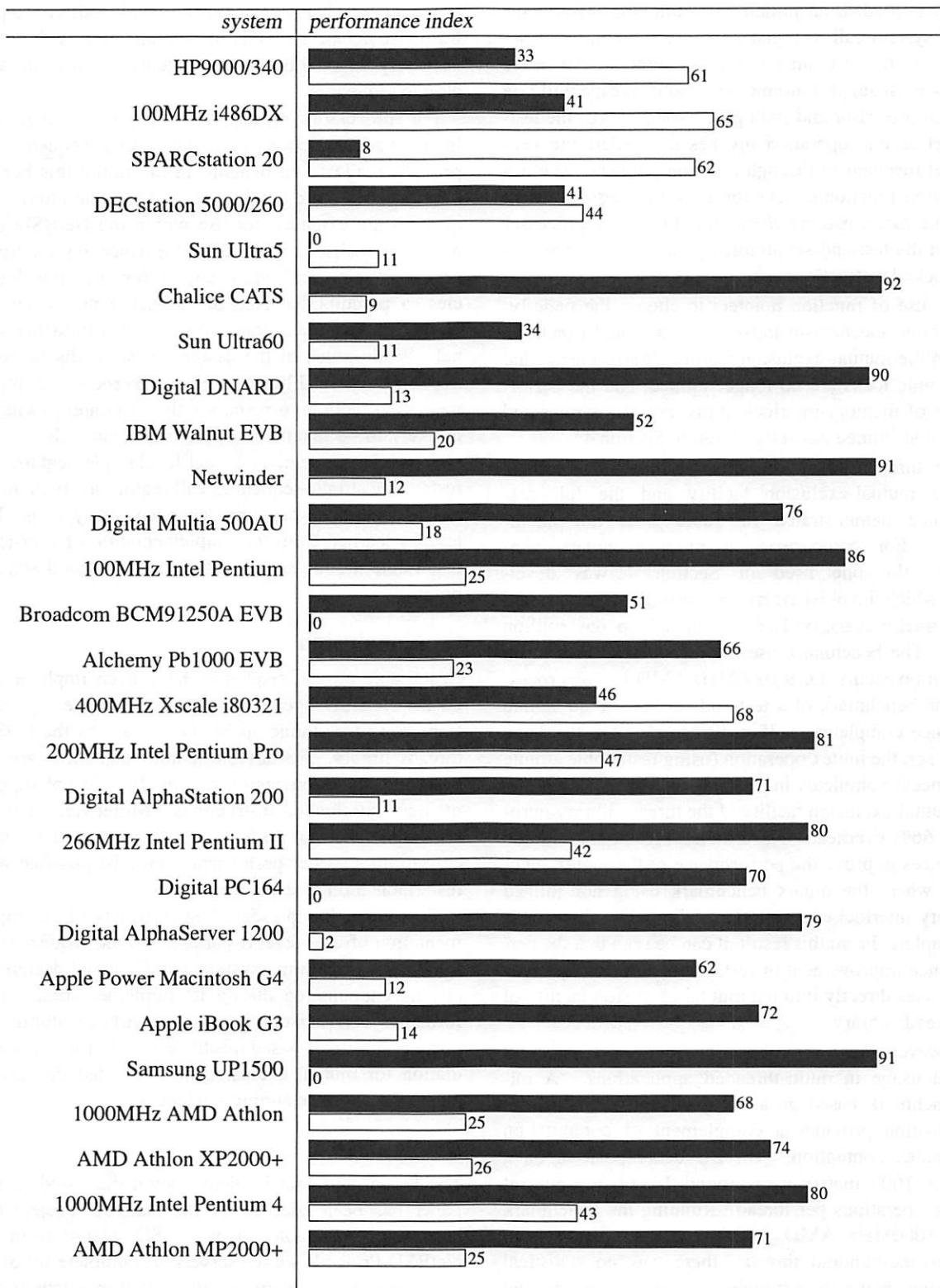


Table 2: Performance index of a test-and-set atomic operation based on restartable atomic sequences (black) and inlined memory-interlocked instructions (white). Larger values of the performance index indicate improved performance over non-inlined memory-interlocked instructions.



a single restartable atomic sequence. When the threads library is loaded, an initialisation function invokes the *rscctl* system call to register the restartable atomic sequence of the test-and-set atomic operation. Since the library must support atomic operations transparently on both uniprocessor and multiprocessor systems, the test-and-set atomic operation invokes the underlying test-and-set mechanism through function pointers. The initialisation function checks for a multiprocessor system with the `hw.ncpu.sysctl` variable. On a multiprocessor system the test-and-set atomic operation uses memory-interlocked instructions.

The use of function pointers to choose the underlying atomic mechanism introduces additional execution cost in the mutual-exclusion facility. It also means that the atomic locks are no longer inlined, and the performance of memory-interlocked instructions is relegated to the non-inlined case considered in Section 4.

The thread library introduces additional overhead to the mutual-exclusion facility and the full performance demonstrated in Table 2 is not attainable. For comparison, a microbenchmark similar to the one used in Section 4 was developed which invoked `pthread_mutex_lock()` and `pthread_mutex_unlock()` in a loop one million times. The benchmark uses a single thread so that no contention occurs. On a 1000MHz AMD Athlon processor, the benchmark of a test-and-set restartable atomic sequence completes in 75 milliseconds. On the same processor, the mutex operation (using restartable atomic sequences) completes in 125 milliseconds. Therefore, the mutual-exclusion facility of the threads library introduces 66% overhead. Nevertheless, restartable atomic sequences improve the performance of the mutex functions, where the mutex benchmark using non-inlined memory-interlocked instructions takes 135 milliseconds to complete. From this result, it can be seen that the performance improvement of restartable atomic sequences propagates directly into the mutual-exclusion facility of the threads library.

However, these microbenchmarks do not represent typical usage in multi-threaded applications. A microbenchmark based on a row-parallel LU matrix decomposition provides a complement of computation and mutex contention. An LU decomposition on a  $1000 \times 1000$  matrix uses around 14,000 test-and-set atomic operations per thread. Running the benchmark on a 1000MHz AMD Athlon processor using from one to one-hundred threads, there was no statistical difference between restartable atomic sequences and memory-interlocked instructions (*p*-value 0.7). Similarly, a benchmark of the multi-threaded apache web server (version 2.0.44) using the *httperf* HTTP performance measurement tool (version 0.8) also showed no

statistical difference in either the request rate or data-transfer rate. Therefore, it may be concluded, from a performance perspective, that most applications will not be adversely affected by the implementation of restartable atomic sequences.

For processors which do not provide memory-interlocked instructions, restartable atomic sequences do provide a significant benefit. In the future this benefit may become more significant if restartable atomic sequences are extended for use within the NetBSD kernel. To realise improved performance on multiprocessor systems and attain targets for real-time latencies, a preemptable NetBSD kernel would place increased demand on atomic operations within the kernel. While much of the design decisions discussed in Section 3 are readily extended to a kernel-level implementation, actioning restarts only on context switches is likely to be insufficient. Interrupts must also action restarts. Consequently, a kernel-level implementation of restartable atomic sequences will require many more invasive changes to machine-dependent subsystems. The lessons learned from this implementation of user-level restartable atomic sequences serves as a good starting point.

## 6 Conclusion

Restartable atomic sequences have been implemented on the NetBSD operating system to provide a generic framework for atomic operations for use by the POSIX threads library. Restartable atomic sequences are appropriate for uniprocessor systems that do not support memory-interlocked instructions. Moreover, on modern processors that do have hardware support for synchronisation, better performance may be possible with restartable atomic sequences.

This paper has presented an overview of an implementation of user-level restartable atomic sequences on the NetBSD operating system and discussed design decisions encountered during its implementation. Performance comparisons between restartable atomic sequences, a syscall-based mechanism and instruction emulation for mutual exclusion demonstrated the advantages of restartable atomic sequences.

## Availability

The kernel and user implementation discussed in this paper has been adopted by the NetBSD Project and is currently available under a BSD license from the NetBSD Project's source servers. A complete set of regression tools for memory-interlocked instructions and restartable atomic sequences is available within the source tree. The next formal release which will use the implementation will be NetBSD 2.0. Further information about the NetBSD Project can be found on the



Project's web server at [www.netbsd.org](http://www.netbsd.org).

## Acknowledgements

Thanks to Artem Belevich, Allen Briggs, Simon Burge, Martin Husemann, Jason Thorpe, Valeriy Ushakov, Martin Weber, Nathan Williams, and Berndt Wulf for the performance data presented in Tables 1 and 2.

## References

- [1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.
- [2] B. N. Bershad, D. R. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, 1992.
- [3] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.
- [4] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.4BSD operating system*. Addison-Wesley, 1996.
- [5] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.
- [6] N. J. Williams. An implementation of scheduler activations on the NetBSD operating system. In *Proceedings of the 2002 Usenix Annual Technical Conference*, 2002.



# Providing a Linux API on the Scalable K42 Kernel

Jonathan Appavoo  
Computer Science Department  
University of Toronto  
jonathan@eecg.toronto.edu

Marc Auslander, Dilma Da Silva,  
David Edelsohn, Orran Krieger,  
Michal Ostrowski, Bryan Rosenberg,  
Robert W. Wisniewski, Jimi Xenidis  
IBM T. J. Watson Research Center  
k42@watson.ibm.com  
<http://www.research.ibm.com/K42>

## Abstract

K42 is an open-source research kernel targeted for 64-bit cache-coherent multiprocessor systems. It was designed to scale up to multiprocessor systems containing hundreds or thousands of processors and to scale down to perform well on 2- to 4-way multiprocessors. K42's goal was to re-design the core of an operating system, but not an entire application environment. We wanted to use a commonly available interface with a large established code base. Because Linux is open source and widely available, we chose to support its application environment by supporting the Linux API and ABI. There were some interesting complications as well as advantages that arose from K42's structure because our implementation of the Linux application environment was done primarily in user space, had to interface with K42's object-oriented technology, and used fine-grained locking. Other research systems efforts directed at achieving a high degree of scalability and maintainability exhibit similar structural characteristics.

In this paper we present the motivation behind K42, including its goals and overall structure, and describe its system interface. We then focus on the required infrastructure and mechanisms needed to efficiently support a Linux application environment. We examine the lessons learned of what was advantageous and what was disadvantageous from K42 in implementing the Linux API and ABI.

## 1 Introduction

The K42 project[6] is developing a new open-source operating system kernel incorporating innovative mechanisms and policies and modern programming technologies. Our goal is to start from a "clean slate" and examine the system structure needed to achieve excellent performance in a scalable, maintainable, and extensible system. Although we wanted to design from scratch, we could not, nor did we want to, implement all aspects of an operating system from scratch. Further, requiring applications to use a new API would make experiment-

ing with the system unpalatable to potential users. We therefore did not introduce a new personality, but instead made K42 Linux API- and ABI-compatible. This paper examines what we needed to do to achieve this compatibility.

K42 has a set of design features that made implementing this compatibility an interesting task. Specifically, in keeping with the design strategy of K42, the API was implemented mostly in user space, had to interface with K42 object-oriented technology, could not use any global locks, and could not hold a lock across multiple object calls. As in many areas of system design, these features both simplified and complicated the implementation. Other scalable and maintainable systems exhibit similar characteristics. Throughout the paper we point out experiences where K42's features impacted (both positively and negatively) the implementation of the API.

To make this approach of supporting the Linux application environment effective, K42 needs to fully support the Linux API and ABI. There is no porting to K42. Any application that runs on Linux just runs if using K42's ABI support and just needs to be re-compiled to use K42's API support. Most applications, including significant benchmarks, run without recompilation. If the application does not run, we fix K42 until it does. We will describe K42 in more detail later, but currently K42 can run significant Linux applications. For example, we have run the SPEC SDET[2] benchmark suite, an Apache web server, and the full ASCI Nuclear Transport Code[21].

Our model for mapping a Linux environment onto K42 is illustrated in Figure 1. A Linux Application Environment, as defined by any of the popular distributions, consists of several layers and modes of operation and execution. As with most Unix-like operating systems, Linux can be segmented into a user level and a kernel level. The user level is the portion of Linux that interacts directly with the user processes, employing the services offered by the kernel level. The kernel level presents the

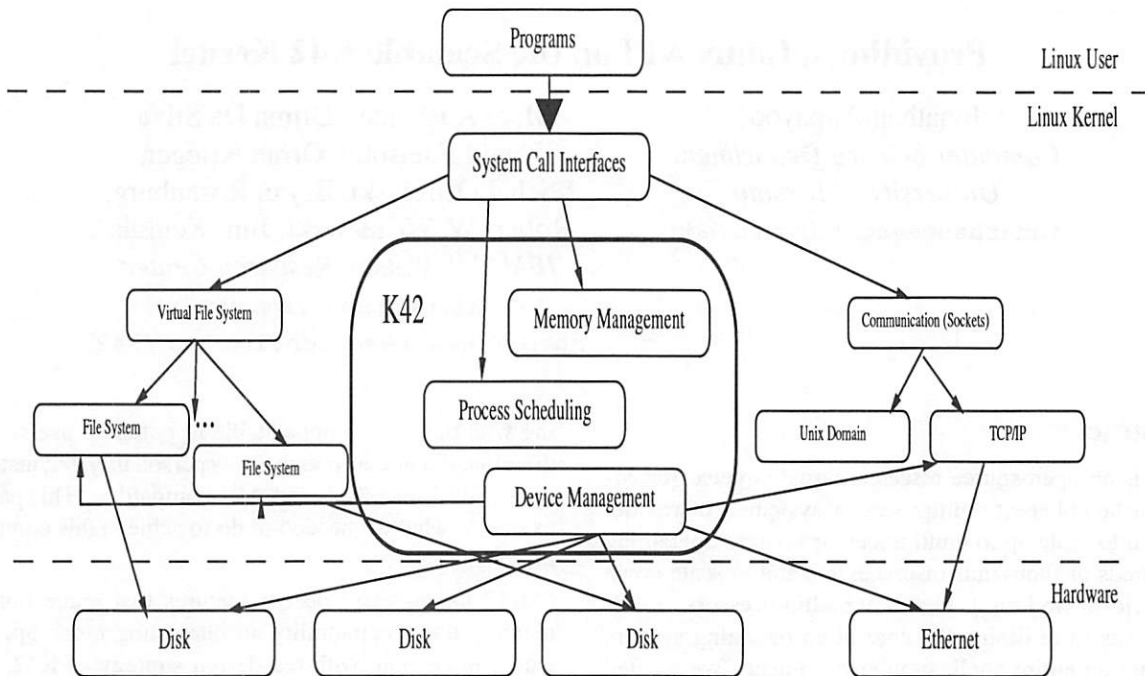


Figure 1: Mapping a Linux environment on K42

image of a single system to each user as well as each application run by that user. It is responsible for managing all resources and securely sharing them.

In order to provide a complete Linux application environment, K42 must provide a set of interfaces that deliver all system services. These come in the form of library functions, system calls, user commands, special or device files, and file formats. Though some of these services do not require kernel support, many of them expose functionality exported by the kernel.

The focus of this paper is on how we provide a Linux application environment on top of K42, a kernel designed for scalability and extensibility. As noted, there are many services that need to be provided in order to present the user with a full application environment. In this paper we examine the major categories of these services and describe how we implement them. In particular, we describe how we emulate the Linux process tree; emulate `fork`, `exec`, and `clone`; support glibc and translate its system calls, provide file systems and sockets, and implement dynamic linking via `ld.so`.

In addition to supporting a full user environment, we wanted to be able to use the existing Linux-kernel code base to provide the desired range of hardware drivers in K42. Although this paper focuses on the user application environment, we briefly outline our kernel strategy here (more details can be found in Auslander et. al.[8]). We use the Linux code base for hardware driver support, for networking and file-system code, and to provide a stable inter-operable environment. To be

able to use Linux device-driver, networking, and file-system code we needed to provide a Linux-kernel environment (or Linux emulation environment, as in Goel and Duchamp[14]). To do so, K42 presents itself as a target hardware architecture for Linux (in the same way as real hardware architectures such as Alpha, i386, and PowerPC do). This requires implementing the basic functionality required of architecture-specific code in Linux (e.g., assembly-level constructs, definition of locking mechanisms) in an “emulation layer” that can be linked with the individual Linux-kernel components to be used in K42. For example, device-driver code uses locks; these locks need to be mapped onto K42 locks and thus the Linux device-driver code needs to be compiled to run in the K42 environment[8]. Further, K42 emulates the Linux-kernel services (e.g., kernel memory allocation) needed to support these components. As seen in Figure 1, this means that K42 replaces the core memory management, process management, and device management code of Linux, but uses file systems, device driver, and library (e.g., glibc) code from Linux.

The rest of this paper is organized as follows. Section 2 starts by introducing K42, describing its motivation and structure, and presenting its system interface. Although much of the API implementation is interrelated, we divided it into mechanisms external to the process, presented in Section 3, and internal to the process, presented in Section 4. Throughout these sections we present the advantages gained from K42’s infrastructure as well as the complications that arose. In Section 5 we



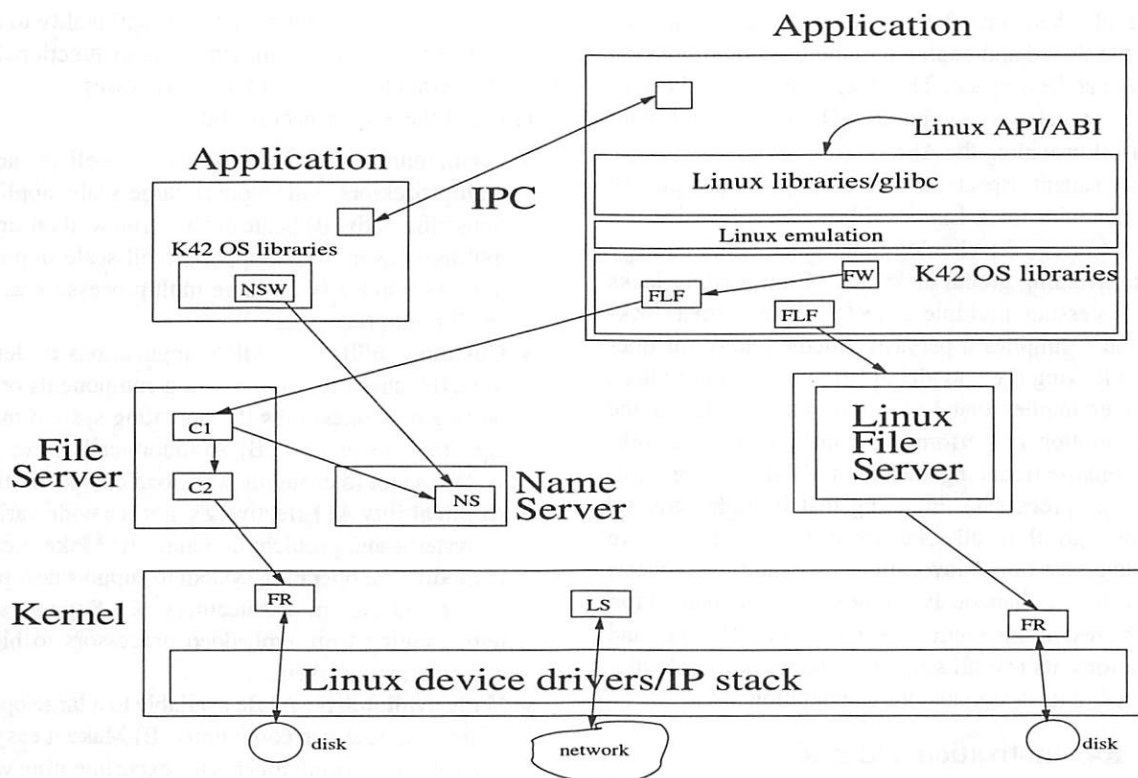


Figure 2: Structural Overview of K42: FR - File Representative Object, LS - Linux Socket Object, C1 C2 - File-system Component 1 2, NS - Name Server Object, NSW - Name Server Wrapper Object, FLF - File Linux File Object

describe the status of K42, provide a performance evaluation of K42 and Linux running the SPEC SDET benchmark, and present work related to K42. Concluding remarks are presented in Section 6.

## 2 K42

In this section we provide background and motivation for K42 and describe its structure and system interface. One of the primary differences between K42's Linux API and Linux is that in K42 much of the standard kernel functionality has been implemented in user space. Results from the Exokernel project[13] demonstrated that user-level implementation of operating system services can lead to significant performance gains. We have found that there are also some performance challenges (more details later, e.g., Section 3.2), especially when implementing a pre-determined model, e.g., `fork`. The Exokernel work showed that moving code into the application's own address space improves performance. Micro-kernel designs did not show performance improvements. K42 is more like the Exokernel.

Although performance was our primary motivation for user-level implementation of kernel services, there are other benefits. Code and data for services implemented in user space do not tie up kernel resources.

User-space code can yield a cleaner programming model because issues such as stack violations are easier to protect against. In user space, allocating virtual stacks, versus pinned stacks in the kernel, allows quasi-infinite stacks with red pages guarding the end. Moving code into the application's own address space does not introduce protection issues. The application still only has access to resources that the prior Linux permissions would have allowed. Moving code into user-level servers introduces a potential scheduling challenge that we address by running the servers at a higher priority than applications.

K42 has addressed security in a first-class manner. All IPC calls between applications, servers, and the kernel contain a badge that securely identifies the caller and is guaranteed by the kernel. For each object invocation on the callee side there is a series of matched rights the badge is compared with to ensure the caller has the proper authentication to make the call. The infrastructure in K42 allows for different security models between different servers and different applications as desired.

In K42, all thread scheduling is done by a user-level scheduler and requests that would normally block in the kernel, e.g., page fault waiting for disk I/O, do not block in the kernel but are instead returned, with control, to the

user-level scheduler. The user-level scheduler can then block the thread and continue running another thread in the same address space. This positively affects the handling of signals, asynchronous I/O, sockets, and other aspects of providing the API.

Other salient aspects of K42's design impacting the implementation of a Linux API are the pervasive use of object-oriented technology and our locking strategy of both avoiding global locks and of not holding locks while accessing multiple objects. The former locking strategy implies a pervasive methodology for fine-grained locking (i.e., no global kernel lock as in Linux). The latter implies that locks may not be held for the entire duration of performing a task, such as a fork-chain collapse (reducing the length of the tree representing forked processes), implying that in-flight requests must be algorithmically accounted for, i.e., if locks are not held across object invocations then multiple requests may occur simultaneously and need to be accounted for.

In the rest of this section we describe K42's goals and motivations, its overall structure, and the key technologies used in its design and implementation.

## 2.1 K42 motivation and goals

K42 focuses on achieving good performance and scalability, providing a customizable and maintainable system, and being accessible to a large community through an open source development model. Supporting the Linux API and ABI makes K42 available to a wide base of application programmers, and our modular structure makes the system accessible to the community of developers who wish to experiment with kernel innovations. K42 is available under an LGPL license (see <http://www.research.ibm.com/K42>).

The system is fully functional for 64-bit applications and currently runs on PowerPC (SMP) platforms (hardware and simulator) and is being ported to x86-64. It runs codes ranging from scientific applications, such as the Splash Benchmark Suite[23] and a full ASCII Nuclear Transport Code[21], to complex benchmarks like SPEC SDET[2] to significant subsystems like Apache.

Providing a well-structured kernel is a primary goal of the K42 project, but performance is also a central concern. Some research operating system projects have taken particular philosophies and have followed them rigorously to extremes in order to fully examine their implications. Although we follow a set of design philosophies in K42, we are willing to make compromises for the sake of performance. The principles that guide our design include 1) structuring the system using modular, object-oriented code, 2) designing the system to scale to very large shared-memory multiprocessors, 3) leveraging performance advantages of 64-bit processors, 4) avoiding centralized code paths, global data structures,

and global locks, 5) moving system functionality to application libraries, and 6) moving system functionality from the kernel to user-level server processes.

Goals of the K42 project include:

- **Performance:** A) Scale up to run well on large multiprocessors and support large-scale applications efficiently. B) Scale down to run well on small multiprocessors. C) Support small-scale applications as efficiently on large multiprocessors as on small multiprocessors.
- **Customizability:** A) Allow applications to determine (by choosing from existing components or by writing new ones) how the operating system manages their resources. B) Autonomically have the system adapt to changing workload characteristics.
- **Applicability:** A) Effectively support a wide variety of systems and problem domains. B) Make it easy to modify the operating system to support new processor and system architectures. C) Support systems ranging from embedded processors to high-end enterprise servers.
- **Wide availability:** A) Be available to a large open-source and research community. B) Make it easy to add specialized components for experimenting with policies and implementation strategies. C) Open up for experimentation parts of the system that are traditionally accessible only to experts.

## 2.2 K42 structure

K42 is structured around a client-server model (see Figure 2). The kernel is one of the core servers, currently providing memory management, process management, inter-process communication (IPC) infrastructure, base scheduling, networking, device support, etc. (In the future we plan to move networking and device support into user-mode servers).

Above the kernel are applications and system servers, including the NFS file server, name space server, socket server, pty server, and pipe server. For flexibility, and to avoid IPC overhead, we implement as much functionality as possible in application-level libraries. For example, all thread scheduling is done by a user-level scheduler linked into each process.

All layers of K42, the kernel, system servers, and user-level libraries, make extensive use of object-oriented technology. All IPC is between objects in the client and server address spaces. We use a *stub compiler* with decorations (additional keywords) on the C++ class declarations to automatically generate IPC calls from a client to a server. The kernel provides the basic IPC transport and attaches sufficient information for the server to provide authentication on those calls, including specific identification of the client being granted access. We have optimized the IPC path as described in Gamsa

et. al. [12] and obtained good performance via efficient IPCs similar to L4[20].

From an application's perspective, K42 supports the Linux API and ABI. This is accomplished by an emulation layer that implements Linux system calls by method invocations on K42 objects. When writing an application to run on K42, it is possible to program to the Linux API or directly to the native K42 interfaces. All applications, including servers, are free to reach past the Linux interfaces and call the K42 interfaces directly. Programming against the native interfaces allows the application to take advantage of K42 optimizations.

The translation of standard Linux system calls is done by intercepting glibc system calls and directing them to their K42 implementation, as described in Section 4.1. Although Linux is the first and currently only personality we support, the base facilities of K42 were designed to be personality-independent. As mentioned in the introduction K42 also supports a Linux-kernel *internal personality* allowing us to use the large code base of drivers, networking, and file-system code.

### 2.3 K42 key technologies

To achieve the above mentioned goals, we have incorporated many technologies into K42. We have written several white papers (available on our web site) describing these technologies in greater detail. This section provides an overview of the key technologies used in K42. At the beginning of this section we highlighted the ones impacting our Linux API implementation.

- Object-oriented technology has been applied to the entire system. This has been used to achieve good performance through customization, to achieve good MP performance by increasing locality, to increase maintainability by isolating modifications, and to perform autonomic functions by allowing components to be hot swapped[24].
- Much traditional kernel functionality is implemented in libraries in the application's own address space, providing a large degree of customizability and reducing overhead by avoiding crossing address space boundaries to invoke system services.
- A structure that permits machine specific features such as the PowerPC inverted page table and the MIPS software-controlled TLB to be exploited in an isolated manner without compromising portability.
- System functionality implemented in user-level servers with good performance maintained via efficient IPCs similar to L4[20].
- The use of *processor-specific* memory (the same virtual address on different processors maps to different physical addresses) to achieve good scalable NUMA performance. This technology, combined

with avoiding global data, global code paths, and global locks, allows K42's design to scale to thousands of processors.

- A K42 specific object-oriented structure, *clustered objects*[12], which provide an infrastructure to implement scalable services with the degree of distribution transparent to the client. This also facilitates autonomic multiprocessor computing[4] as K42 can dynamically swap between uniprocessor and multiprocessor clustered objects.
- K42 is designed to run on 64-bit architectures and we have taken advantage of 64 bits to make performance gains by, for example, using large virtually sparse arrays rather than hash tables.
- K42 is fully preemptable and most of the kernel data is pageable.
- K42 is designed to support a simultaneous mix of time-shared, real-time, and fine-grained gang-scheduled applications.
- K42 has developed deferred object deletion [12] similar to RCU [22] allowing objects to release their locks before calling other objects. This efficient programming model is crucial for multiprocessor performance and is similar to type-safe memory [16].

## 3 Linux process external environment

The next two sections describe K42's implementation of the Linux API. Much of the code and therefore description is interrelated. This section presents the infrastructure and code mostly external to a Linux process.

### 3.1 The Linux process tree

In order to implement a Linux environment the Linux process tree must be supported. In K42 this functionality is provided by the ProcessLinuxServer. This server was implemented primarily by using Linux data structures, such as the `task_struct`, with only minor changes to track Linux processes in K42.

Linux processes are backed by native K42 processes and contain a pointer to the underlying K42 process. In addition to backing Linux processes, K42 processes are used to implement core facilities. Linux processes are used for user applications and many of K42's servers, such as the filesystem. Should additional personalities be added to K42, they would add additional types of processes.

The ProcessLinuxServer keeps track of the relationship between different Linux processes. It tracks the parent-child relationship when a process is created via `fork` (we only intend to support forking of Linux processes). When a parent process finishes or is terminated, the ProcessLinuxServer re-assigns the child to `init`. In addition to the process tree, the ProcessLinuxServer



keeps track of two other trees or sets. It tracks the sessions used to associate processes that are related by their starting tty. It allows any process to get access to the session tty even though the parent has not passed the child an open fd for it. In addition to sessions, the ProcessLinuxServer keeps track of the set of process groups. As in Linux, every process knows what process group and what session it belongs to.

In addition to maintaining the three structures mentioned above, the ProcessLinuxServer manages Unix credentials, and as in Linux, matches process to process groups where appropriate. The ProcessLinuxServer is also responsible for the delivery of signals (discussed in more detail later). The Linux process tree, combined with session and process groups, forms the core tracking infrastructure for many of the services described in the rest of the paper.

### 3.2 Fork

Unlike many implementations of Unix, K42 executes much of the fork code in the client, both in the parent and in the forked child, and does not grab a global tree-lock thereby improving concurrency. Concretely, the parent sets up the memory state of the child. It has a record of all the memory mappings (regions) associated with itself, much the same as the Linux kernel keeps track of all regions associated with a process. After setting up the memory of the child, it passes the rest of the state to the child. In Linux, execution normally starts in the child at the instruction of the `fork` system call. However, in K42, a significant amount of code is executed in the child before branching to the instruction following the `fork`. This code is similar to what would have been executed in the kernel, such as setting up the file descriptor table, etc.

The K42 model of removing this code from the kernel has several advantages. It reduces the amount of code in the kernel. Any errors that occur in it do not crash the system, but only bring down the process involved in the fork. Perhaps most importantly, it yields an easier programming model as the code does not have to be written assuming fixed sized stacks or written to use a limited amount of pinned memory.

Although implementing the code in user space simplifies programming, the K42 requirement of not holding a global lock throughout such an operation requires careful coding to avoid potential race conditions. Another performance advantage, but programming difficulty, is that when performing a fork-chain collapse (reducing the length of the tree representing forked processes), we do not hold a lock across all the operations but rather lock each individual node independently. Thus, the algorithm must allow for the possibility of multiple in-flight requests (remove and insert). This is discussed in greater

detail below.

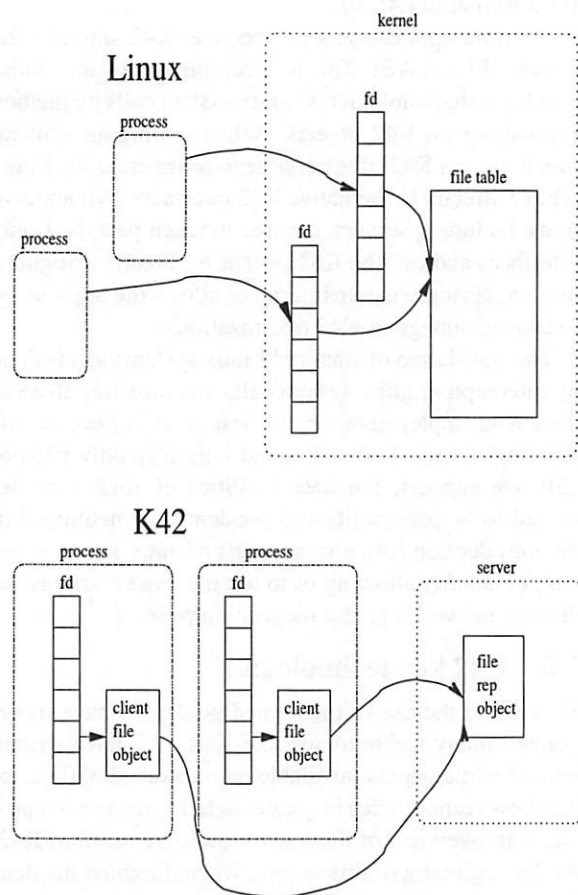


Figure 3: The fd table in Linux and equivalent structure in K42

The object-oriented nature of K42 has serious implications when replicating structures such as the file descriptor (fd) table. In Linux, a fd is a pointer into a table (see Figure 3) with associated file reference counts that are incremented when a `fork` occurs. However, in Linux, the fd table takes kernel memory for each process because it is stored in the kernel's data structure for the process image. In K42, the fd array, which is maintained in user space, points to a series of client file object instances that are in each process's address space (see Figure 3, more information on file descriptor manipulation in K42 is presented in Section 4.5). There is a little more memory that must be replicated on a `fork` for the fd table in K42.

Although K42's approach avoids explicit replication of the fd array, it introduces post-`fork` authentication issues. Each of the object instances contains authentication information providing a particular client the right to access a given file. The authentication information contains a PID and thus, although valid for the parent, does not provide the child with permission to access that same



file. A call on each file object must be made for the child to obtain permission. This could be an expensive operation if the parent had a large number of open files. It is especially unfortunate because it is unlikely the child will access many of those files as its mostly likely action is to `exec`. We avoid this potential performance bottleneck by lazily granting access to the child on first access to the file server.

As is frequently the case in programming large systems, the tradeoffs are not clear, and although some aspects of K42's implementation of `fork` simplified programming or improved performance, other aspects made it more complicated or hurt performance. For K42 native processes (that do not support `fork`) the model of performing much of the process creation code in user space has been a win. Unfortunately, `fork` has difficult performance issues. The overhead of implementing `fork` and `exec` in user level comes from (1) the cost of passing state from parent to child, rather than just copying in kernel, (2) the extra fault overhead to map in data structures the parent passed to child copy-on-write, (3) the cost of initializing (on `exec`) and re-initializing (on `fork`) the system function implemented in user level (e.g., the fd array, memory allocation, etc), and (4) the overhead from our end-to-end authentication scheme (for each file being passed to the child, the server needs to be involved to provide access). Although we bought back most of the fork performance lost due to these issues by lazy (re)initialization, there is still some performance loss and work is underway to reclaim it. Fortunately the performance of `fork` for most server applications (some benchmarks notwithstanding) is not critical.

### 3.3 Reproducing the memory image on a fork

After a `fork`, the memory image of the parent must be reproduced in the child. Any private mappings for a file cause a snapshot of that file at the time of the `fork` to be produced in the child. Any shared mappings cause a shared mapping to be created in the child with modifications made by the parent to be replicated in the child, with the possible caveat of files mapped in read-only but modified by the debugger through the use of `ptrace`.

Traditionally the operation of reproducing these mappings is done in the kernel. However, in K42, before starting the child, the parent reproduces these mappings with the help of a few underlying K42 kernel services. K42 provides the ability to create a new non-executing process, the ability to make memory mappings in it, and the ability for a region and associated objects to be fork copied. In K42, a contiguous piece of memory is represented by a region, which in turn is backed by a File Cache Manager (FCM) that caches the in-core pages,

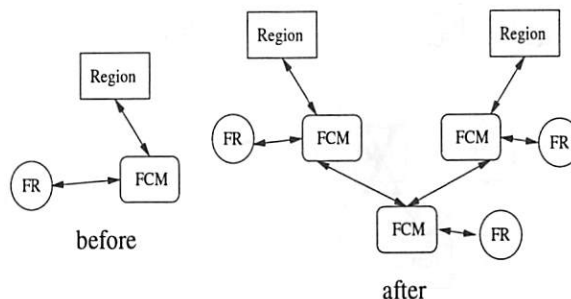


Figure 4: Example of Region, FCM, and FR objects after a fork copy

and a File Representative (FR) that maps all the pages associated with a file. Figure 4 illustrates a standard fork copy. Prior to the fork (*before*), the three kernel structures (FR, FCM, Region) represent a contiguous portion of the memory in the parent. After the fork (*after*), the upper-left three kernel structures still represent the portion of memory in the parent, and the upper-right three structures represent the same portion of memory in the child. All the frames backing the region are immediately transferred to an internal FCM (the root FCM in this figure) and lazily copied back to the parent and child as faults occur.

As additional forks occur, a binary fork-tree is produced. When nodes on the tree are removed because the process for which they are representing memory finishes or is terminated, it is important to “collapse” the tree, otherwise inefficiencies result because page-fault processing needs to traverse the unnecessarily long chain of nodes. The standard way to address this is to acquire a lock and walk the tree performing the collapse. Because K42 is programmed with independent object instances representing each of the regions, and with a programming model not allowing the acquisition of a single lock across multiple object instances, we acquire and release a fine-grain lock for each node in the tree. The FCM objects therefore need to be able to handle in-flight page fault requests occurring during the collapse operation. Although this does make the programming more complicated, fine-grain locking schemes provide better scalability.

The collapse algorithm begins when a node in the tree needs to be removed. That node removes itself and it tells its sibling to combine with their parent. If the sibling is not a leaf node then all frames are transferred to the parent and the sibling node removes itself. If the sibling is a leaf node then no action is taken. To prevent a chain from forming in this case, on the next fork copy, the new child is made a child of the existing parent (no new parent is created). For this algorithm to work, frames are not copied to an internal child node (except in one case for optimization purpose only), instead they are

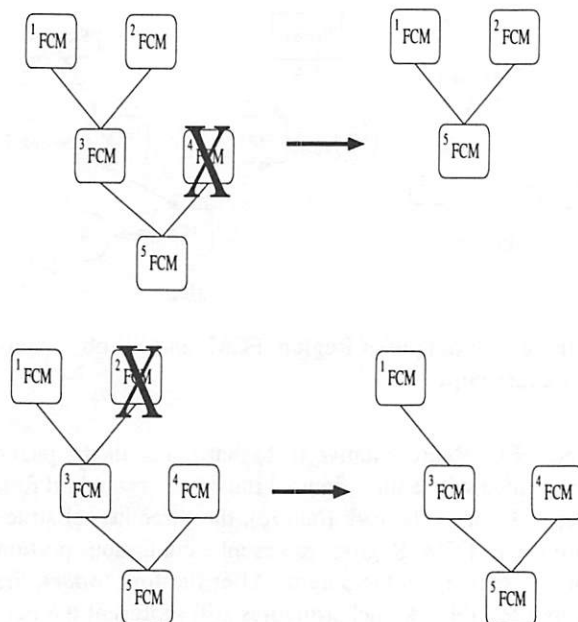


Figure 5: Standard scenarios in fork-chain collapse

copied to the leaf node requesting the page. As examples, two standard scenarios are illustrated in Figure 5. In the first, node 4 is being deleted. This results in a collapse with node 5 becoming the parent of nodes 1 and 2. In the second, node 2 is being deleted. As above, because it is a leaf node, its removal does not induce a collapse.

In the normal *fork/exec* case we do not perform any collapse. The algorithm is in place to allow K42 to continue to perform acceptably in scenarios that would generate long fork-tree chains. This algorithm provides efficient collapsing of fork trees and does not require a global lock.

### 3.4 Reproducing the fd and signal state on a fork

In addition to reproducing the memory in the forked child, the filesystem state and signal state must also be copied. Most of this state is replicated during the reproduction of the memory state. A mechanism is needed to indicate the child is now using the files referenced by the open fds and to receive the signals set up by the handlers.

In Unix, after a *fork*, the kernel indicates the child is using the files by traversing the fd list and incrementing the usage count on each of the files pointed to by that fd list. As mentioned above, in K42 there is an individual object instance that manages each file. User-space FileLinuxFile objects have permission to talk to the underlying file through the objectRef they hold to that file. The vast majority of calls to *fork* are followed by calls to *exec* in the forked child. In K42, we lazily provide

permission to each of the files in the child. In the common case of a succeeding *exec*, we therefore avoid the work altogether. In Unix, this would be equivalent to not actually updating the reference count until an access is made (but also being careful to not decrement the count if no reference is made). In K42 we accomplish this by creating an object in the kernel to represent the client file being reproduced in the child. In a fork intensive workload, where many files are not accessed after fork, we pay the same overhead as a kernel implementation like Linux (because the parent pushes this information to the kernel), but, when files start being used, the K42 kernel interacts with the server to perform a proper setup (the child lazily initializes the structures), and from then on we get the advantages of our user-level implementation.

The signal state of the parent is copied to the child entirely by the memory reproduction. Thus, all signal handlers installed in the parent are reproduced in the child. The disposition of pending signals due to, for example asynchronous I/O, is undefined. In K42 the signal will be delivered only to the old object that was originally waiting for it.

### 3.5 Signals

Signal delivery in K42's Linux application environment differs from most implementations in that most operations can be carried out without kernel involvement. State information such as signal masks, *sigaction* specifications (ignore, default action, or handler to call when signal arrives), and set of pending signals, is kept in the application's address space. Operations such as *sigsuspend* and *pause* simply block in the client without any interaction with the kernel.

Inter-process signal delivery goes through the ProcessLinuxServer. The target K42 process is identified, and in the general case, the server delivers the signal to the client through an IPC. For signals that can not be ignored (*SIGKILL*, *SIGSTOP*), the server contacts the kernel.

Intra-process signal delivery is carried out entirely in the client space. This results in significant performance advantages for the current implementation of Linux Threads[19], which uses signals to implement synchronization among threads. In the upcoming Linux threads package(NPTL)[10], synchronization is implemented using the *futex* (fast light-weight user-space semaphores) kernel synchronization service. Although *futexes* are more lightweight than the current implementation, they still involve kernel interaction. With the new threading model, we still expect to be able to implement the synchronization at user level.

## 4 Linux process internal environment

### 4.1 Linux system calls and glibc

The largest component of the Linux API/ABI that must be supported is the set of system call interfaces. For true ABI compatibility this must be supported by performing system call trap reflections from the code making the Linux system call to the implementation of the system call. In K42, the functions that implement system calls reside within `exec.so`, a pre-loaded library within the process described in Section 4.2.

Support for the trap reflection mechanism is intended for strict compatibility. We have designed but not yet implemented this mechanism. In general, we expect applications running on K42 to use a version of the GNU C Library (glibc) targeted for K42 to obtain access to the Linux system call interface without using trap reflection. Most applications gain access to Linux system call interfaces via glibc wrapper functions that present a C interface thereby hiding the architecture specific details of making a system call. Glibc provides a mechanism that allows different architecture targets to define how these wrapper functions work. This is accomplished by providing the assembly-level mechanisms for making system calls. Except for this low-level assembly system call interface level, K42 reuses all of glibc. We have developed a glibc targeted for K42 that efficiently accesses the implementations of system calls within a process. We use the system call number as an index into a *system-call transfer-table*. This table is at a well-known location and contains the address of the function that implements a particular system call. In effect, we short-circuit the normal trap-reflection and simply jump to the correct function.

The system-call transfer-table permits us to handle Linux system calls within an application, but at the same time avoids the need for us to explicitly link applications against K42 code. As a result, dynamically linked PowerPC64 Linux binaries that use glibc wrappers to make system calls run without modification. Applications linked against a static non-K42 glibc still require trap reflection.

As previously noted, in K42 we have moved kernel functionality into the application's address space. This includes some of the processing for system calls. Moving this code into the application's address space does not sacrifice security. The portion of the system call that is handled in the user's address space is the part that is allowed by the Linux credentials the user had. The model enforces calling privileged servers to perform requests where the user credentials are insufficient. Under either model, users can cause incorrect behavior (e.g., if an application has read/write access to a file one thread may remove the file another thread is attempting to write).

This model does introduce additional places where the user may "shoot themselves in the foot", but it does not sacrifice security.

When a thread makes a Linux system call, a wrapper object in our Linux emulation layer is invoked. At this point, the thread is marked as being in a system call. This code is re-entrant allowing additional calls from within the process. If a signal is delivered to a thread in this state, it is deferred until the return from the outermost level of system call. The thread will not block when it is in this state, but instead will return out of the system call layer and then block[7].

### 4.2 Loading

K42 provides a library called `exec.so`. This library provides the OS functionality that K42 expects to be implemented within a process. It also provides the system-call transfer-table and the functions it points to. `Exec.so` maintains data such as stateful I/O interfaces that implement select and poll, and must be initialized before any Linux code executes. This early initialization needs to occur so that we can provide support for system calls made by the dynamic linker and glibc.

A K42 process can be created using native K42 facilities (not `fork/exec`). This creation occurs by constructing the objects that represent that new process in the parent. During this construction, the parent loads `exec.so` into this new or *child* process. It also loads information containing what executable that child process should run. The new child process starts running in `exec.so` and loads the appropriate Linux executable, and if necessary, the dynamic linker. As part of its initialization, `exec.so` creates and initializes the system-call transfer-table, and then jumps to the appropriate entry point of the Linux application. The Linux application is now able to make system calls that are handled by the code in `exec.so` via the system-call transfer-table, or by trap-reflection, which also uses the table. This mechanism is established without any symbol dependencies between the Linux application and `exec.so`.

We must also provide a mechanism to allow applications to access K42 library interfaces directly. However, we cannot simply allow the dynamic linker to load a new image of `exec.so` because the existing K42 code in `exec.so` contains important state information. Instead, the library image that the dynamic linker loads contains specially modified ELF headers that direct the dynamic linker to look for K42 library code, data, and symbol information in the location where `exec.so` has been already loaded. In a 64-bit address space it is easy to always load `exec.so` at the same location allowing us to avoid the need to relocate `exec.so` at run-time.



### 4.3 Exec

The `exec` system call conceptually creates a new process based on a set of predefined specifications. The memory image of the new process is clean, files not marked close on `exec` are maintained open in the new process, and signals not set to ignore are reset to default.

Like other services in K42, `exec` is implemented primarily in user space. Given the above model for loading, a K42 process may be considered an `exec.so` image that loads or overlays Linux executables when asked to `exec`. With this model, the same K42 process containing `exec.so`, can, when asked to perform an `exec`, unload the Linux executable and any shared libraries it has loaded, re-initialize the appropriate aspects of the Linux personality, and load the new Linux executable. In the common case, `exec` can be performed entirely in user space.

It is not possible to use the overlay model if the `exec` is for a `setuid` program (or for `clone` or native K42 processes). Because the old program could have stored permissions not granted to the new process anywhere in its memory image, we have a privileged server create a clean process, and copy the necessary state. This server uses the underlying K42 mechanisms used by `fork`.

The overlay strategy for implementing `exec` permits us to efficiently handle most Linux applications and perform the `exec` in user space. For the cases where we can not use the overlay strategy, most operations are still performed in user space.

### 4.4 Clone

The `clone` system call creates a new process like `fork` does, but it offers more control of which elements in the execution context (memory space, table of file descriptors, table of signal handlers, file system information, process id, etc) are intended to be shared between the child and parent processes. The main use of `clone` is to implement `LinuxThreads`[19], which are multiple threads of control in a program that run concurrently in a shared memory space.

The implementation of `clone` is carried out completely in the application address space, i.e., no interactions with the kernel or servers are necessary. `clone` builds on K42's user-level thread model[7]. Even though the kernel does not allocate these thread PIDs we still need a mechanism to guarantee unique PIDs across the system. We do this by reserving a bit range in the PID space for `clone` PIDs. PIDs are 32 bit in Linux. K42 uses 20 bits to identify the target K42 process and 12 bits to identify the `clone` within the process. This allows `clone` PIDs to be assigned locally by each process and still ensure that `clone` PIDs are globally unique.

### 4.5 Files, file descriptors, and name space

Usual implementations of UNIX APIs store client (application) specific state information regarding IO/IPC (files, sockets, pipes, etc) elements in the kernel. For example, for files, the kernel not only manages attributes such as ownership and file length, but also is responsible for keeping state information (e.g. file position) for each client. File descriptors *fds* (offsets in a per-process table kept by the kernel) are provided as an argument by the clients to identify the target of the operation. Most operations on such elements may need to block. If so, the blocking is also carried out in the kernel.

K42 takes a different approach. The client-specific information for open files, sockets, and pipes resides in the application's address space. K42 stores information for stateful interfaces in the object implementing that interface, which is placed in the application's address space. When blocking is necessary, the thread will block until notified by the server that the operation can proceed. As a consequence, kernel resources (kernel thread stack, process state, register state, etc.) are not tied up. This avoids the difficulties encountered in *m on n* scheduling models. In some scenarios, the operations can execute entirely in the application space, thereby achieving significant performance gains.

Operations such as `socket()`, `pipe()`, `creat()`, and `open()` create a new IO/IPC element, returning a file descriptor for it. K42's implementation of these system calls does the following: (1) the client contacts the name-space server (or `MountPointServer`) to discover the appropriate server for that resource, (2) the client initiates an interaction with the appropriate server. The server identifies the server object representing the resource (creating one, if necessary), checks credentials, and returns to the client a handle for the server object. This handle includes capabilities indicating that the client is allowed to invoke the server object directly, and (3) the client creates an object to represent the client side of the open file, socket, or pipe, storing in it the object handle to the server object. A file descriptor is associated with the newly created client object, and this mapping is stored in a file descriptor array kept by the application.

Subsequent operations on the file descriptor will be delegated to the client object. The client object implementation usually invokes the corresponding operation on the server object representing the resource, and updates its own local information. In some cases, the client object is able to carry out the operation completely. For example, for a small file with a single client, file position, file data, and all `stat` information (including file length), can be managed in the client. The information is propagated to the server when the file is closed, becomes too large to be reasonably cached in the client, or is accessed by additional clients.



K42's synchronous I/O model is similar to other asynchronous models in the sense that threads do not block in the kernel or servers. To illustrate how blocking and unblocking occurs in the application space even for synchronous requests, we describe our implementation of sockets. All socket interfaces in K42 are similar to Unix ones, except there is an extra parameter to pass back to the client information about the state of the object (for example, whether it is readable, writable, or if there is any exceptional condition on it). The client uses this state information to decide if it should block. The server makes an asynchronous upcall to notify the client of state changes (e.g. data becomes available). This scheme allows us to implement `select` and `poll` purely in user space. Also, asynchronous notifications are piggybacked on synchronous responses to client invocations. Our implementation is also able to detect when a client is ignoring notifications (for example, a forked child that inherited the file descriptor but is not interested in the socket), and to stop sending them.

Many interfaces such as signals, `select` and `poll`, and cursor management in files, require synchronization. Linux provides this synchronization in the kernel. K42 either provides this in the application space (if that application is the only user of the resource), or in the server object managing that resource. In fact, we have an interesting example of the hot-swapping mechanism[24] that is used to switch between these two implementations when the requests for a given resource change from coming from a single application to originating from multiple applications.

K42's support for namespace traversal involved in pathname-based operations (similar to Welch and Ousterhout[25]) has performance and scalability advantages over the usual pathname lookup schemes due to its fine-grained locking and ability to resolve in the application address space the parts of the pathname that identify mount points. A `MountPointServer` stores the association between parts of the name space and specific file-system servers. The information available in this server is cached in the application's address space during its initialization phase. The `MountPointServer` publishes the version number for its up-to-date information. An application can use these version numbers to check efficiently if it has out-of-date information.

In the uncommon case that the information is not up-to-date, the application requests the current information from the `MountPointServer`. Once the mounting information is used to resolve part of the pathname and identify the corresponding file-system server, the client contacts the file-system server and passes arguments of the operation to be performed and the unresolved part of the pathname.

The file-system independent layer in K42 implements

caching of directory and file entries recently resolved. Each file-system instance has its own caching data structure. Fine-grained locking is used when manipulating this data structure, avoiding well-known scalability bottlenecks in name resolution.

## 5 Status, Performance, and Related Work

In this section we describe the status of K42, describe work related to K42, and finish with a performance evaluation of K42 and Linux running the SPEC SDET benchmark[2].

K42 is available under an LGPL license and a CVS (Concurrent Version System) source tree is available. Directions on how to obtain it are available at <http://www.research.ibm.com/K42>. An early version of a complete environment including build infrastructure, debug tools, a simulator, and source is available as of March 2003.

The modular structure of the system makes it a good teaching, research, and prototyping vehicle. Policies and implementations studied in this framework have been transferred into Linux. For example, a kernel scalable queue lock originally designed in K42 was transferred to Linux, RCU[22] is similar to the safe memory and garbage collection in K42, and lockless scalable tracing technology has been integrated into LTT[26]. K42's framework will allow continuing technology transfer.

K42 currently runs on PowerPC (SMP) hardware and simulators (SimOS and Mambo), and is being ported to x86-64. As stated, K42 is fully functional for 64-bit applications, and can run codes ranging from scientific applications to complex benchmarks like SDET to significant subsystems like Apache. Currently, K42 can directly execute 64-bit binaries compiled for Linux, and soon will be able to do the same for 32-bit binaries.

There are still some missing holes in K42's full Linux compatibility. There are system calls with unimplemented (less common) cases. For example, `mmap` protection only works on common cases. We have not yet implemented `/proc` (except for `ps`, this is primarily an impediment only to running administration tools). Our approach has been to add additional Linux functionality as applications require it. In some sense, K42 will never be fully one hundred percent compatible. We do not intend to be bug compatible, in fact, stack overflows and other such error conditions should they occur would be at different places in K42 than in Linux.

Although Linux is currently the only personality K42 supports, the base K42 mechanisms would allow support of other interfaces as well. Other flavors of Unix such as AIX and BSD would be fairly easy to support. Most of the challenging technical work is already in place. There would still be considerable detailed work to be done in ensuring structures get correctly translated. For

example, getting the exact semantics and contents of the `stat` structure correct on such systems can be difficult because the structure is different under each of the Unix systems mentioned above. Other potential challenges involve providing dynamic linking for systems like AIX that use XCOFF instead of ELF. Supporting a significantly different interface, e.g., Windows, while doable, would be considerably more work. K42 is not much further along being able to support a Windows API than Linux is. The original intent in K42 was to allow multiple personalities to be efficiently supported, but we have not pursued this avenue of research. Other work[18] has examined supporting multiple personalities on a micro-kernel-like architecture.

Other operating systems emulate Linux, for example, all the BSD Unix systems do. They do so by reflecting traps to a vector with minor translation from Linux to native system calls. That is one part of our solution, the more straightforward part. The more difficult part for K42 is that it is not a real Unix system under the covers. Thus, one of the key challenges relate to the different abstractions the base system supports. Mach is another operating system with an implementation of Linux. The MkLinux Linux Server [1] has the entire Linux functionality in one single Mach task (instead of smaller specialized tasks communicating through Mach RPCs) in order to maximize reuse of the existing monolithic kernel.

K42 has similarities to a micro-kernel such as Mach[3] but more closely resembles the Exokernel[11]. The kernel, filesystem, etc., servers do not provide all the operating services, rather part of this functionality is integrated into the application's own address space. For the Linux API, a large majority of the conversion occurs in each application's address space. This approach is unique to K42.

There are other operating system projects that have aspects similar to K42. The ability to perform efficient IPCs as in L4[20] is important to K42's structure of employing user-level servers. K42's strategy for fault tolerance is similar to Disco[15] in that the plan is to run multiple simultaneous instances of K42 across varying sized machines. Other operating systems such as Spring[17] and Choices[9] have similar object-oriented goals but were motivated more by distributed system concerns. Our approach is best summarized by what was stated earlier, namely that performance was a central concern and although we follow a set of design philosophies in K42, we are willing to make compromises for the sake of performance.

## 5.1 Performance

K42 has been designed to achieve scalable performance. To date, this has been our primary focus. More recently, we have started to tune uniprocessor performance. The

goal of our K42 design is to achieve near perfect scalability while still maintaining uniprocessor performance very close to that of other operating systems. Moreover, as Linux makes additions for multiprocessor performance, K42 should be able to match or better Linux's uniprocessor performance through our use of specialization and hot-swapping[5][24].

In this section we describe the SPEC SDET benchmark[2] and its performance on both K42 and Linux. The experiment shows that Linux out-performs K42 on a uniprocessor (we continue to work to reduce this gap), but that K42 significantly outperforms Linux on a medium size (24-way) multiprocessor.

The Standard Performance Evaluation Corporation (SPEC) Software Development Environment Throughput (SDET) benchmark consists of a script that executes a series common Unix commands and programs including `ls`, `nroff`, `gcc`, `grep`, etc. Due to missing infrastructure, for our experiments (both K42 and Linux), the SDET benchmark was modified by removing the system utilities `ps` and `df`. Each of the commands in the script are run in sequence. To examine scalability we ran one script per processor. We ran the same script on both K42 and Linux 2.4.19 as distributed by SuSE with the O(1) scheduler patch. All the user programs (`bash`, `gcc`, `ls`, etc.) are the exact same binary. The same version of `glibc` 2.2.5 was used, but modified on K42 to intercept and direct the system calls to the K42 implementations. The experiments were run on an S85 Enterprise Server IBM RS/6000 PowerPC bus-based cache-coherent multiprocessors with 24 600MHZ RS64-IV processors and 16GB of main memory.

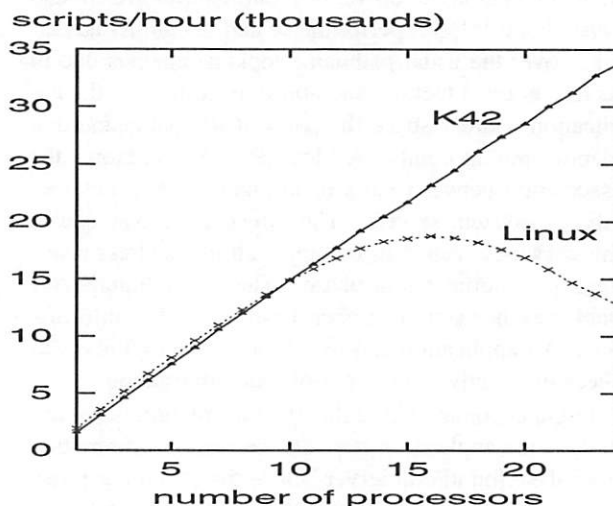


Figure 6: Results of running SDET on K42 and Linux on a 24-way multiprocessor

Figure 6 shows the results from the experiment. The script is timed and results are reported in thousands of

scripts per hour. On a uniprocessor, Linux achieves 1905.2 scripts/hour and K42 runs 1575.0. K42 suffers about a twenty percent performance degradation on a uniprocessor. Linux reaches a peak of 18749.0 at 16 processors and by 24 processors executes at a rate of 12710.7 scripts/hour. K42's performance surpasses Linux by 10 processors, at which point K42 executes 14912.6 scripts/hour while Linux executes 14856.0 scripts/hour. K42 continues to scale well through 24 processors where its peak of 33808.1 scripts/hour is achieved yielding an efficiency of 89.4 percent. These results demonstrate the effectiveness of K42's scaling.

We continue to work on our scaling to increase our efficiency to 100 percent. There is also continuing work to help Linux to scale better, and 2.6 is expected to demonstrate better scalable performance. Concurrently, we are working on K42's uniprocessor performance.

One of the advantages of K42 is the object-oriented model and the resulting well-modularized structure allowing well-contained coding experiments to be implemented. We hope to see an increase in interest in using K42 for a rapid prototyping tool as well as a platform to pursue scalable and first-class customization research. Recently, there has been an increase in interest from academic collaborators looking to use K42 as a base to pursue research, and we look forward to continuing to support increased activity with K42.

## 6 Conclusions

K42 is a new open-source research operating system kernel designed from the ground up for scalable cache-coherent 64-bit multiprocessor systems. To provide access to a wide community and code base, we implemented mechanisms to support a Linux API and ABI. The desire to have a high performance, scalable, and maintainable operating system has resulted in several interesting features impacting our implementation of the Linux application environment on top of K42. These features include the implementation of kernel services in user space, object-oriented technology, avoidance of global locks, and avoidance of locking across calls to multiple objects. These characteristics in whole or in part will be representative of future systems that are designed to be scalable and maintainable. We described our experiences and lessons learned where these features impacted the implementation of the Linux API.

K42 is under active development at IBM T. J. Watson, and collaborating Universities. Interested parties may check out the project at: <http://www.research.ibm.com/K42>.

## Acknowledgments

Thank you to the referees and especially our shepherd, Ray Bryant, for their careful reading of this document

and their helpful suggestions. This work was supported by IBM's ASSR (Adventurous Systems and Software Research) and SSI (Server Systems Institute) initiatives.

## References

- [1] Mklinux. <http://www.ota.be/linux/workshops/19970607/Generale/>.
- [2] SPEC SDM suite. <http://www.spec.org/osg/sdm91/>, 1996.
- [3] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevastian, and M. Young. Mach: A new kernel foundation for UNIX development. In *USENIX*, pages 93–112, July 1986.
- [4] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma da Silva, Orran Krieger, Marc Auslander, David Edelsohn, Ben Gamsa, Gregory R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenberg, Michael Stumm, and Jimi Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76, 2003.
- [5] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert Wisniewski, Dilma da Silva, Orran Krieger, and Craig Soules. An infrastructure for multiprocessor run-time adaptation. In *WOSS - Workshop on Self-Healing Systems*, 2002.
- [6] Marc Auslander, David Edelsohn, Dilma da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenberg, Robert W. Wisniewski, and Jimi Xenidis. *K42 Overview*. IBM Research, <http://www.research.ibm.com/K42>, August 2002.
- [7] Marc Auslander, David Edelsohn, Dilma da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenberg, Robert W. Wisniewski, and Jimi Xenidis. *Scheduling in K42*. IBM Research, <http://www.research.ibm.com/K42>, August 2002.
- [8] Marc Auslander, David Edelsohn, Dilma da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenberg, Robert W. Wisniewski, and Jimi Xenidis. *Utilizing Linux Kernel Components in K42*. IBM Research, <http://www.research.ibm.com/K42>, August 2002.
- [9] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, September 1993.
- [10] Ulrich Drepper and Ingo Molnar. The new native posix thread library for linux. <http://people.redhat.com/drepper/nptl-design.pdf>, October 2002.
- [11] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *ACM Symposium on Operating System Principles*, volume 29, 3–6 December 1995.
- [12] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Symposium on Operating Systems Design and Implementation*, pages 87–100, 22–25 February 1999.
- [13] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceno, Russell Hunt, and Thomas Pinkney. Fast and flexible application-level networking on exokernel systems. *ACM Transactions on Computer Systems*, 20(1):49–83, February 2002.
- [14] Shantanu Goel and Dan Duchamp. Linux device driver emulation in mach. In *USENIX Annual Technical Conference*, pages 65–74, 1996.
- [15] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 18(3):229–262, 2000.
- [16] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 123–136. ACM Press, 1996.

- [17] Graham Hamilton and Panos Kougiouris. The spring nucleus: A microkernel for objects. In *Summer USENIX Conference*, pages 147–160, June 1993.
- [18] Freeman L. Rawson III. Experience with the development of a microkernel-based, multi-server operating system. In *HotOS - Workshop on Hot Topics in Operating Systems*, pages 2–7, 1997.
- [19] Xavier Leroy. The linuxthreads library. <http://pauillac.inria.fr/~xleroy/linuxthreads/>.
- [20] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250. ACM Press, 1995.
- [21] Mark M. Mathis, Nancy M. Amato, and Marvin L. Adams. A general performance model for parallel sweeps on orthogonal grids for particle transport calculations. In *ICS Proceedings of the 14th ACM International Conference on Supercomputing*, pages 255–263, May 2000.
- [22] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Proceedings of the Ottawa Linux Symposium*, 26–29 June 2002.
- [23] J. P. Singh, W.-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *CAN*, 20(1):pp. 5–44, March 1992.
- [24] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenberg, and Jimi Xenidis. System support for online reconfiguration. In *USENIX*, page to appear, San Antonio, TX, June 2003.
- [25] B. Welch and J. K. Ousterhout. Prefix tables: A simple mechanism for locating files in a distributed system. In *Proc. IEEE Int. Conf. on Distributed Computing Systems*, pages 184–189, 1986.
- [26] Karim Yaghmour. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.



# THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

## SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

## Member Benefits

- Free subscription to *login*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *login* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

## USENIX & SAGE Thank Their Supporting Members

### USENIX Supporting Members

- ❖ Ajava Systems, Inc. ❖
- ❖ Aptitune Corporation ❖ Atos Origin B.V. ❖
- ❖ Computer Measurement Group ❖
- ❖ Freshwater Software ❖ Interhack Corporation ❖
- ❖ The Measurement Factory ❖
- ❖ Microsoft Research ❖ Sun Microsystems, Inc. ❖
- ❖ UUNET Technologies, Inc. ❖
- ❖ Veritas Software ❖ Ximian, Inc. ❖

### SAGE Supporting Members

- ❖ Certainty Solutions ❖ Freshwater Software ❖
- ❖ Microsoft Research ❖ Ripe NCC ❖

For more information about membership, conferences, or publications,  
see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA  
Phone: 510-528-8649 Fax: 510-548-5738 Email: [office@usenix.org](mailto:office@usenix.org)

ISBN 1-931971-11-0